MACHINE LANGUAGE ROUTINES FOR THE COMMODORE 64/128

Todd D. Heimarck and Patrick Parrish

A comprehensive collection of more than 200 machine language routines for the Commodore 128 and 64, ready to add to your programs. Includes routines to access printers, disk drives, and Kernal routines; sorting algorithms; and much more. The ideal reference.

A **COMPUTE! Books** Publication $18.95

Machine Language Routines
for the Commodore 64 and 128

Todd D. Heimarck and Patrick Parrish

COMPUTE! Publications, Inc. abc
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-085-8

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64 and 128 are trademarks of Commodore Electronics Limited.

Contents

Preface

This book is a rich library of more than 200 machine language routines for programmers to learn from and use in their own programs. The programs in this book cover a wide range:

- Character input and output
- Sprite definition and movement
- High-resolution graphics
- Sorting and searching lists of information
- Reading and writing disk files
- Combining BASIC and ML programs
- Printer routines
- Addition, subtraction, multiplication, and division
- Conversions between character and screen codes
- Random number generation
- Jiffy clock and time-of-day clock routines
- Using interrupts and vectors
- Custom characters (for 40- and 80-column displays)
- Sound effects and music

These are just a few of the routines you'll find in this book. Nearly every subroutine is listed with a sample program that illustrates how it works. You can study the subroutine by itself or see how it's used in the context of a real program.

One of the best ways to learn machine language is to study other people's programs. If you can see how someone else got the computer to do something like moving sprites, printing a score, sorting a list, or whatever, you can trace through the steps and gain a better understanding of the technique.

But most magazines and books publish machine language (ML) as a series of numbers in DATA statements. You don't learn much about machine language from typing in clusters of numbers. You could use an ML monitor to disassemble the program, but when you're faced with a sea of JSRs and BEQs, it's not always obvious what's going on in a program.

The programs include a wealth of comments that take you step by step through the various stages of each routine: setting up the variables, calling the routine, and handling the results.

Most routines are written for the Commodore 64, but will run on the 128 with the changes indicated in comments. A few routines will work only on the 64 or only on the 128, but most will run on both computers.

In addition to the 200-plus routines, we've included a complete list of ML opcodes, with explanations of how they work, and a complete list, with explanations, of the built-in Kernal routines.

Whether you're a machine language beginner or a seasoned expert, we think you'll find many useful programming techniques, routines, and ideas in this book.

Todd D. Heimarck
Patrick Parrish

> All the source code in this book is ready to type in and assemble. There is also a disk available from COMPUTE! Books which includes all the source code from the book (no object code is included on the disk). An assembler is required to use the disk. To purchase the disk, use the coupon in the back of the book or call 1-800-346-6767 (in New York 212-887-8525).

Introduction

The paradox of machine language is that it's both simpler and more complex than a high-level language such as BASIC or Pascal.

Machine language (ML) is simpler because a program consists of many very small steps. LDA #10 puts the number 10 in the accumulator. STX $C115 takes the number in the X register and stores it in memory location $C115. If you study a single line from a fast and powerful machine language program, you'll usually see that not very much happens. Now consider a BASIC command such as SPRITE on the Commodore 128. With one command, you can turn a sprite on; set its color, priority, and expansion; and put it in multicolor mode. Compared to the Spartan instruction set of ML, BASIC is a richer and more complicated language.

But even though the instructions are small and simple, putting together a working ML program is often more complex than writing a BASIC program. If you make a mistake, chances are good that the program will go into an endless loop (or worse, the computer will crash). There are no convenient error messages to tell you what you did wrong. You're responsible for keeping track of your own variables. And you're expected to understand some of the architecture of the computer—how the various support chips and their registers work.

Some people find ML quite easy. Others struggle to learn it. Either way, we hope you'll discover some useful routines in these pages.

What You'll Find Here

This book is divided into three major parts: the instruction set, the Kernal routines, and the machine language routines.

The instruction set lists each 6502 machine language operation, with an explanation of what it does and which flags are affected. The 6502 family of chips includes the 6510 in the 64; the 8502 in the 128; and the 6502 in the VIC-20, Atari 400/800, and original Apple II. The ML instructions listed are common to all of these computers. (Incidentally, if you pro-

gram on these other 6502-based computers, you may be able to translate some of the routines in this book for the VIC, Atari, or Apple.) The instruction set contains the building blocks of ML programming. If you're a beginner, you may want to look through this section first. Even if you're an old pro, you'll need to refer to this list occasionally.

The next section of this book—"ROM Kernal Routines"—lists the Kernal routines (which are common to all eight-bit Commodore computers, including the VIC, Plus/4, 64, and 128). Note the deliberate misspelling of what other computer manufacturers call *kernel* routines. The Kernal is a block of memory that uses a standard jump table to make it easier to program on different brands of Commodore computers. For example, the routine that prints a character is found at different locations on the 64 and 128, but the standard entry point for the Kernal CHROUT routine is the same ($FFD2) on both computers. This means the line LDA #65: JSR $FFD2 will work the same on both computers—it prints the letter *A* on the screen. Indeed, it also works on the VIC-20, the Plus/4, and the 16. We're indebted to Ottis Cowper for giving us permission to reprint a portion of his *Mapping the Commodore 128* (COMPUTE! Books, 1986) that explains how the Kernal routines work.

The importance of the Kernal routines cannot be overemphasized. To open a disk file, you call the Kernal routines SETLFS, SETNAM, and OPEN. (See the entries under **OPENFL** or **READFL** for examples.) If these routines weren't available, it would be quite difficult to read from or write to a disk file; you'd have to write your own disk operating system, with routines to spin the disk, move the read/write head to a given sector, read bytes one at a time, and so on.

The third and largest part of the book is the collection of ML routines. Each subroutine is listed alphabetically by label. In some cases, the entire program is the subroutine. However, the routine is usually put in the context of a framing program which illustrates how to set up and call the given subroutine (marked by bold type). When a routine appears elsewhere in the book, its label appears in **boldface** type.

What You Won't Find Here

The book is big, but we couldn't include everything. One thing you won't find is an explanation of how to begin programming in ML. If you're a beginner, you'll find useful

examples and programs here, but you may also want to look into two books for beginners: *Machine Language for Beginners* by Richard Mansfield (COMPUTE! Books) and *Machine Language* by Jim Butterfield (Brady Books). Mansfield's book takes a software approach, relating machine language instructions to their BASIC counterparts. If you know how a FOR-NEXT loop works in BASIC, this book shows you how to do the same thing in ML. Butterfield's book approaches ML more from the hardware viewpoint, explaining what's happening inside the computer while an ML program is running. We highly recommend both books.

When you're writing programs for the 64 and 128, it's necessary to understand something about how memory is organized—which zero-page locations are available; which ROM routines are useful; how the registers of the support chips control video, input/output, and sound. For a general introduction to these topics, Commodore's two programmer's reference guides are excellent. The 64 version is published by Howard Sams; the 128 version comes from Bantam Books. For more detail, *Mapping the Commodore 64* by Sheldon Leemon and *Mapping the Commodore 128* by Ottis Cowper are essential (both published by COMPUTE! Books). In fact, if you buy only one other machine language book, get the mapping book for your computer. We also recommend *Anatomy of the Commodore 64* and *128 BASIC 7.0 Internals* (Abacus Books). Both books feature commented disassemblies of the BASIC ROMs.

The Routines

Each machine language routine has a label up to six letters long. Following the label is a more descriptive name that tells you what the routine does, for example, **SQROOT**: Calculate the integer square root of an integer value.

Below the label and name, you'll see one or two paragraphs that touch on the main points of the routine, with examples of where you might use the routine or a summary of how it works.

Next is the prototype, which is something like a flowchart converted to instructions written in English. It lists the individual steps followed by the subroutine and points out the variables and memory used within the routine. There are usually three steps covered in the prototype: how to set it up, how the routine works, and how the results are handled.

Following the prototype is a more in-depth explanation of what the framing program does. This section discusses alternate ways to use the subroutine, more information about how to modify it for your own purposes, how certain tasks were accomplished, how memory is affected, and so on. Often there's an important note or even a warning. The **FORMAT** routine formats a disk, for example, which warrants a warning that if you run this program, you'll erase everything on your disk.

Finally, the source code for the program is listed. Some routines are a few lines, others cover several pages. We recommend that you use a symbolic multipass assembler to type in these programs (see below). Although you can use a monitor such as Micromon or Supermon, you'll find that an assembler is preferable.

Typing In and Assembling the Programs

We chose the *Personal Assembly Language (PAL)* assembler to write the source code for the routines in this book. The 64 version (*PAL*) and 128 version (*Buddy-128*) are available from many Commodore software dealers, or from the distributor, Pro-Line Software in Mississauga, Ontario. If you use the LADS assembler from *The Second Book of Machine Language* (COMPUTE! Books), you'll find that the source files are mostly compatible, with very minor changes.

If you're using another assembler, such as Commodore's *Macro Assembler Development System* (MADS), Eastern House Software's *Macro Assembler/Editor* (MAE), Roger Wagner's *Merlin*, or one of the others available, you may need to make a few modifications to get the source code to run.

First, a note about pseudo-ops. The three letters LDA represent a machine language instruction (or operation). The mnemonic LDA is translated to a number that's POKEd into memory or saved in a disk file by the assembler. The operation LDA is always followed by one or two bytes that provide additional information. These bytes are the operand. In the instruction LDA $C150, LDA is the operation, and $C150 is the operand. The assembler converts this line to the numbers 173, 80, and 193 ($AD, $50, and $C1). For this instruction and addressing mode, LDA is the mnemonic, and $AD is the equivalent opcode.

Assemblers usually include additional commands that aren't really part of the ML instruction set, but they're instruc-

tions to the assembler. For example, *PAL* takes .OPT OO to mean "Object where Origined," or "assemble this to memory." *Buddy-128* uses .MEM. LADS uses .O. These pseudo-operations tell the assembler to do one thing or another.

At the beginning of most programs, you'll see a series of *equates*, each of which instructs the assembler to assign a label to a memory location. The memory location may be the entry point for a Kernal ROM routine, it may be a location in RAM, or it may be a register in the VIC or CIA or SID chip. One of the most common equates looks like this: CHROUT = $FFD2. This informs the assembler that the label CHROUT, when encountered later in the program should be replaced by the address $FFD2. JSR CHROUT means JSR $FFD2. Some assemblers use the pseudo-instruction EQU in place of the equal sign. If your assembler follows this convention, instead of CHROUT = $FFD2, you'd substitute CHROUT EQU $FFD2. If you're using a machine language monitor or an assembler that doesn't allow labels, you'll have to make the substitution yourself. The source code may look like this:

$C020 20 D2 FF JSR CHROUT

With Micromon or Supermon, you'd have to look to the left at the D2 FF and translate the instruction (in your head) to JSR $FFD2. Note that the low byte precedes the high byte in the object code to the left.

Both *PAL* and LADS support the #< and #> pseudo-operations. From a two-byte address, the first (#<) extracts the low byte, and the second (#>) extracts the high byte. So if a previous equate assigned the memory location $902F to the label NAMES, the line LDA #<NAMES tells the assembler to load the accumulator with the low byte of NAMES. Since NAMES is $902F, this is equivalent to LDA #$2F. If you saw LDA #>NAMES, it would be the same as LDA #$90. Again, you can look to the left to find the value being referenced.

Some other pseudo-ops include .BYTE, .WORD, and .ASC. If you see a line like ZEBRA .BYTE 15, it means that the byte value of 15 is inserted in the program at the given location and that particular memory location is given the label ZEBRA. Some assemblers use DB (Data Byte) instead of .BYTE. The .WORD pseudo-op translates a two-byte quantity to its low byte and high byte. The .ASC is followed by a quotation

mark and a series of one or more characters, which are stored in memory as Commodore ASCII values.

If you don't understand an instruction that contains a pseudo-op, look to the left for the equivalent object code.

Using the Routines in Your Own Programs

The programs in this book have all been tested. The original source code was assembled and printed to disk (using *PAL*'s .OPT P option), and then uploaded directly to the computer used to typeset this book. So as far as we know, there are no typographical errors in the program listings.

But that doesn't mean that each routine is perfect and ready to be inserted as is in your own programs. For one thing, nearly all of the example programs start at $C000 (decimal 49152). At the very least, you'll probably want to relocate the routines to other parts of memory, especially if you're using a 128. You should also watch for conflicts among routines that use zero-page locations. Many routines depend on indirect-Y addressing and locations 251–252 and 253–254 ($FB–$FC and $FD–$FE). In some cases, you'll have to substitute other available zero-page addresses.

Many of the routines were written to be general and flexible solutions to a problem. If you have a more specific application in mind, you might want to dispense with the subroutine and insert a modified version of a routine directly in your main program. You may also see ways to shorten a routine or make it run faster. We encourage you to experiment with the programs.

For 128 Users

Since most of the programs call Kernal routines, you'll need to be in bank 15, where addresses $0000–$3FFF are RAM in bank 1 and $4000–$FFFF appear as ROM. Instead of assembling programs to $C000, try $0C00 (decimal 3072) on the 128. To take full advantage of the 128K of memory, you need to understand how the different memory banks are accessed. Both the *128 Programmer's Reference Guide* and *Mapping the Commodore 128* discuss how to switch between banks.

About the Disk

A companion disk that contains all the routines in this book is available for purchase from COMPUTE! Books. The programs

are included as source code, not object code, which means you'll need an assembler like *PAL* or LADS to create the runnable program—the object code.

The source files take up much more space than is available on a single-sided 1541 disk, so both sides were used. The disk is a flippy: To use the first half of the programs, use one side; to load the other programs, flip the disk over. The original source files filled more than the 1328 blocks available on a flippy. Rather than omit programs from the disk, we chose to abbreviate the comments in a few programs. Thus, the comments in the source code on disk may not be exactly the same as the comments in the listings in this book. If you list the programs on the disk, you may find that *hi byte* has replaced the phrase *high byte* in the book, for example.

Opcodes

Opcodes

ADC

ADd with Carry: Add a value to the accumulator, with the result in .A.

Addressing Modes

(Zero page,X)	ADC ($FC,X)	61 FC	6 cycles
Zero page	ADC $FA	65 FA	3 cycles
Immediate	ADC #$45	69 45	2 cycles
Absolute	ADC $10	6D 10 00	4 cycles
(Zero page),Y	ADC ($FB),Y	71 FB	5 cycles (+1 over a page)
Zero page,X	ADC $03,X	75 03	4 cycles
Absolute,Y	ADC $A401,Y	79 01 A4	4 cycles (+1 over a page)
Absolute,X	ADC $C002,X	7D 02 C0	4 cycles (+1 over a page)

Flags

N (Negative)	If the result is $80–$FF, the N flag is set.
V (Overflow)	If an overflow occurs, V is set.
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If the result is zero, Z is set.
C (Carry)	If the result exceeds $FF, C is set.

ADC starts with the number in the accumulator and adds to it the given value (which varies according to which addressing mode is used), *plus an additional 0 or 1, depending on the state of the carry flag.* Remember to clear the carry flag (CLC) before addition is started. If you're adding large numbers (two bytes or more), the carry bit will take care of itself. As the addition progresses toward higher bytes in the number, the carry bit spills over into the next most significant byte. When you're adding multiple bytes, add together the least significant first—the low byte—and proceed to add the more significant bytes later.

The carry flag is set when two bytes are being added (say, 250 and 10) and the total is more than can be stored in one

byte (more than 255). If you're in binary-coded decimal mode (D flag set to 1) when addition occurs, the carry flag is set if the sum of two bytes exceeds 99.

The result of addition is found in the accumulator. If you want to save this number, be sure to STA after the addition.

AND

Bitwise AND: Perform a bitwise AND between .A and a value. Result resides in .A.

Addressing Modes

(Zero page,X)	AND ($E6,X)	21 E6	6 cycles
Zero page	AND $22	25 22	3 cycles
Immediate	AND #$1B	29 1B	2 cycles
Absolute	AND $1E5C	2D 5C 1E	4 cycles
(Zero page),Y	AND ($F9),Y	31 F9	5 cycles
Zero page,X	AND $50,X	35 50	4 cycles
Absolute,Y	AND $C493,Y	39 93 C4	4 cycles (+1 over a page)
Absolute,X	AND $3BC3,X	3D C3 3B	4 cycles (+1 over a page)

Flags

N (Negative)	If bit 7 is set, N flag is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If result is zero, Z is set.
C (Carry)	—

AND performs a bitwise AND. Corresponding bits in .A and the value are compared; if either bit is off, the result is zero. Both bits must be on for the resulting bit to be set.

In the example, bits 0, 6, and 7 of the second value ($3E) are off, so the effect is that those bits are cleared from the original number ($AB). To turn bits on, use ORA.

```
    $AB 1010 1011
AND $3E 0011 1110
    -------------
    $2A 0010 1010
```

ASL

Arithmetic Shift Left: Shift a value (accumulator or memory) to the left.

Addressing Modes

Zero page	ASL $4F	06 4F	5 cycles
Accumulator	ASL	0A	2 cycles
Absolute	ASL $DF01	0E 01 DF	6 cycles
Zero page,X	ASL $EF,X	16 EF	6 cycles
Absolute,X	ASL $AA05,X	1E 05 AA	7 cycles

Flags

N (Negative)	Bit 6 shifts into 7 and sets/clears the N flag.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If bits 0–6 are zero, Z is set.
C (Carry)	Bit 7 shifts into carry.

ASL causes all eight bits to shift one position to the left. A zero is placed into bit 0, while bit 7 moves into the carry flag. In contrast, an ROL instruction does the same thing *except* that ROL rotates the carry flag into bit 0. With ASL, a zero is always put into bit 0.

ASL is often used to double a number, to test bits with the N or C flag and branch accordingly, or to perform a two-byte shift. When a two-byte shift is being carried out, ASL is used with ROL; you ASL the low byte and ROL the high byte.

BCC

Branch if Carry Clear: Branch forward or backward if the C flag is clear.

Addressing Modes

Relative	BCC $12B4	90 A5	2 cycles (+1 over a page)

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

BCC operates off the carry flag, which is affected most often by addition and subtraction (ADC and SBC) and by compares

(CMP, CPX, CPY). As with the other branch operations, the range is limited to 127 bytes forward or 128 bytes backward.

After ADC, a cleared carry means that there is *no* carry to be concerned about. After SBC, a cleared carry means there *is* a borrow to handle.

A compare instruction leaves the carry bit in one of two states: If the number in the register is larger than (or equal to) the value being compared, carry is set. If the register is smaller, carry is clear. So LDX #$05: CPX $6793: BCC will cause the branch to happen if the number in .X is smaller than the number at $6793. If $6793 holds a number between $06 and $FF, the BCC will branch to the given address.

BCS

Branch if Carry Set: Branch forward or backward if the C flag is set.

Addressing Modes

Relative	BCS $4578	B0 B2	2 cycles (+1 over a page, +1 if branch occurs)

Flags

N	(Negative)	—
V	(Overflow)	—
-		—
B	(Break)	—
D	(Decimal)	—
I	(Interrupt)	—
Z	(Zero)	—
C	(Carry)	—

BCS, like its counterpart BCC, works off the carry flag. It is seen most often after addition or subtraction operations (ADC, SBC) or after compares (CMP, CPX, CPY). As with the other branching instructions, the range of the branch is limited to 127 bytes forward or 128 bytes backward.

After ADC, a set carry indicates that the result of the addition has exceeded the size of a single byte—in other words, the result is greater than 255. After SBC, a set carry means that no borrow has been necessary (the result is between 0 and 255).

Following compares, carry may be set or cleared. If the number in the register is larger than (or equal to) the value being compared, carry is set. Otherwise, carry is cleared (mean-

ing the value in the register is smaller). So, LDA $FB: CMP #$0A: BCS will cause branching to a given address to occur if the number in location $FB is greater than or equal to $0A ($0A–$FF).

BEQ

Branch if EQual to zero. Branches forward or backward if the Z flag is set.

Addressing Modes

Relative	BEQ $CE9A	F0 10	2 cycles (+1 over a page)

Flags

N	(Negative)	—
V	(Overflow)	—
-		—
B	(Break)	—
D	(Decimal)	—
I	(Interrupt)	—
Z	(Zero)	—
C	(Carry)	—

BEQ can branch up to 127 bytes forward or 128 bytes back. Although most assemblers allow you to specify a target address or label, the address is not assembled. Instead, an offset is calculated (numbers $00–$7F are forward branches; $80–$FF are backward).

There are two ways in which the Z flag may be set. After a load instruction (LDA, LDX, LDY), Z is set if the value loaded is zero. Other instructions (transfers, addition, and so forth) may also affect the Z flag. In this case, the BEQ takes effect if the result is a zero.

After a compare (CMP, CPX, CPY), the Z flag is set if the register and value compared are equal. Here the BEQ means "branch if the two numbers compared are equal."

BIT

Test memory BITs: AND the accumulator with memory, without storing the result.

Addressing Modes

Zero page	BIT $04	24 04	3 cycles
Absolute	BIT $DC01	2C 01 DC	4 cycles

Flags

N (Negative)	Bit 7 of memory is copied to N.
V (Overflow)	Bit 6 of memory is copied to V.
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If the result of the AND is zero, Z is set.
C (Carry)	—

The BIT instruction performs a bitwise AND between the accumulator and a specified memory byte. (See the entry under AND for an explanation and example of a bitwise AND.) The zero flag is set or cleared as a result of the AND. Unlike the AND instruction, which alters the value in .A, BIT affects only the status register. The accumulator remains intact after BIT.

Within the status register, bits 6 and 7 take on the corresponding bit values of the specified memory byte. When testing these bits, BIT is generally followed by BVC/BVS or BMI/BPL, causing the appropriate branch.

BIT instructions are frequently placed in succession at the beginning of a subroutine. Entering the routine at different points causes the status flags to take on different values. But more significantly, the address following each BIT may actually be used as an opcode. This allows you to load different values into a register (A, X, or Y) or to carry out other operations, depending upon the entry point.

For example, say you have a subroutine where you want the value of .Y to start out as $00, $A5, or $B5. You could begin the routine with LDY #$00: BIT $A5A0: BIT $B5A0. If you jump in at the byte following the first BIT instruction, the Y register will load $A5 ($A5A0 is stored low byte first, $A0 $A5, which executes as LDY #$A5). The next BIT instruction will affect only the status register, leaving .Y unchanged. If you jump in at the $B5A0 instruction, an LDY #$B5 will execute and fall through into the subroutine.

BMI

Branch if MInus: Execute a branch if the N flag is set.

Addressing Modes

Relative	BMI $3CA3	30 7B	2 cycles (+1 over a page)

Flags

N (Negative) —
V (Overflow) —
– —
B (Break) —
D (Decimal) —
I (Interrupt) —
Z (Zero) —
C (Carry) —

BMI can branch forward up to 127 bytes or backward, 128. The branch occurs if the N (negative) flag is set. A negative number is one that has bit 7 set and falls in the range $80–$FF. A variety of instructions—adds, subtracts, loads, compares—set the N flag.

BNE

Branch if Not Equal: Branch forward or backward if the Z flag is clear.

Addressing Modes

Relative	BNE $4102	D0 3A	2 cycles (+1 over a page, +1 if branch occurs)

Flags

N (Negative) —
V (Overflow) —
– —
B (Break) —
D (Decimal) —
I (Interrupt) —
Z (Zero) —
C (Carry) —

BNE can branch up to 127 bytes forward or 128 bytes backward. Assemblers generally calculate this offset from a specified target address or label. An offset of $00–$7F indicates a forward branch; $80–$FF, a backward branch.

A branch with BNE takes place when the Z flag is cleared. The zero flag (Z) may be cleared several ways. It's set if the result of an operation is zero; it's cleared if the result is not equal to zero. After a load instruction (LDA, LDX, LDY), Z is cleared if the value is nonzero. Tranfers, addition, and other instructions affect the Z flag similarly. In these cases, BNE causes a branch if the result is not zero.

Following a compare (CMP, CPX, CPY), the Z flag is cleared if the register and value are different. Here the BNE means "branch if the two numbers compared are not equal."

BNE often follows a decrement instruction (DEX, DEY) at the end of a loop. The loop continues its operation as long as the Z flag is cleared.

BPL

Branch if PLus: Branch forward or backward if the negative flag is clear.

Addressing Modes

Relative	BPL $959F	10 DE	2 cycles (+1 over a page)

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

BPL branches if previous instructions have cleared the negative flag. Although you usually specify an address or target, BPL assembles into the instruction plus an offset—forward 0–127 bytes ($00–$7F) or backward 1–128 bytes ($FF–$80).

BPL is commonly used in loops where .X or .Y starts out with a positive value (0–127), and then DEY or DEX counts down to zero. Zero is a positive number, so the BPL loop continues until a final decrement wraps around to $FF, which is negative.

BRK

BReaK: Causes a forced interrupt.

Addressing Modes

Implied	BRK	00	7 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	Set to 1

D (Decimal) —
I (Interrupt) Set to 1
Z (Zero) —
C (Carry) —

BRK halts the ML program, saving the contents of the program counter and the status register (with B and I set) to the stack. Following this, it jumps to the service routine at $FFFE.

The service routine itself points to a routine at $FF48 ($FF17 on the 128), which checks for the B flag. Finding it set, it jumps through the BRK vector at $0316.

Normally, this vector points to a BASIC warm start (on the 64). Many ML monitors, including *Micromon* and *Supermon*, substitute in this vector the address of their own initialization routine, designed to print the contents of the program counter, data, and status registers. When a BRK is encountered, the monitor is enabled, and the current status of the registers is printed. On the 128, the vector points to the built-in machine language monitor.

BVC

Branch if oVerflow Clear: Branch (relative) if the V flag is clear.

Addressing Modes

Relative	BVC $2235	50 64	2 cycles (+1 over a page)

Flags

N (Negative) —
V (Overflow) —
— —
B (Break) —
D (Decimal) —
I (Interrupt) —
Z (Zero) —
C (Carry) —

The V (overflow) flag is important only when you're using signed arithmetic. Since adding $FF to $06 results in $05 (plus a set carry), the number $FF acts like a −1. $FE is −2, $FD is −3, and so on. Within signed arithmetic, the negative numbers include $80–$FF (128 through 255 or −128 through −1), the positive numbers $00–$7F (0–127).

With unsigned arithmetic (numbers 0–255), the carry flag, C, indicates when an overflow has occurred: numbers larger than 256 or smaller than 0. In signed arithmetic (numbers −128 through 127), an overflow happens when the result is larger than 127 or smaller than −128. The V flag is set when there's an overflow from bit 6 to bit 7. BVC enables you to branch forward or backward based on the current state of V.

BVS

Branch if oVerflow Set: Branch (relative) if the V flag is set.

Addressing Modes

Relative	BVS $B1DE	70 9F	2 cycles (+1 over a page, +1 if branch occurs)

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

BVS acts on a set overflow (V) flag, branching as many as 127 bytes forward or 128 backward.

The V flag is used primarily for work in signed arithmetic (with numbers ranging from −128 through 127). Here, bit 7 holds the sign of the number. Positive values run from $00 through $7F (0 through 127); negative numbers from $80 through $FF (128 through 255 or −128 through −1).

Prior to the addition or subtraction of two signed numbers, V is usually cleared with CLV. If overflow occurs from bit 6 to 7 as a result of the operation, it means a number larger than 127 or smaller than −128 has been generated. The V flag is set to indicate that a sign change has occurred. A BVS instruction, which generally follows, will then direct the program to branch accordingly.

BVS is also used after BIT when bit 6 of a specified value is being tested.

CLC

CLear Carry: Clear the carry flag.

Addressing Modes

Implied	CLC	**18**	2 cycles

Flags

N	(Negative)	—
V	(Overflow)	—
-		—
B	(Break)	—
D	(Decimal)	—
I	(Interrupt)	—
Z	(Zero)	—
C	(Carry)	Sets C to zero.

CLC clears the carry flag, which is necessary for the ADC (ADd with Carry) instruction to work properly. It may also be used to force a branch. In the absence of a branch-always instruction, CLC: BCC will suffice. The carry flag also affects rotates (ROL and ROR).

CLD

CLear Decimal mode: Turns off binary-coded decimal (BCD) mode.

Addressing Modes

Implied	CLD	**D8**	2 cycles

Flags

N	(Negative)	—
V	(Overflow)	—
-		—
B	(Break)	—
D	(Decimal)	Set to zero.
I	(Interrupt)	—
Z	(Zero)	—
C	(Carry)	—

CLD is used to restore the computer to its normal binary mode, typically after some BCD operation has been performed.

While decimal mode is in effect (entered with SED), bytes can range in value from 0 through 99, and nybbles from 0 through 9. To carry out a decimal calculation, execute an SED, do the math, and restore binary mode with CLD.

CLI

CLear Interrupt flag: Reenable maskable (IRQ) interrupts.

Addressing Modes

Implied	CLI	58	2 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	Sets I to zero.
Z (Zero)	—
C (Carry)	—

Interrupt requests (IRQs) occur 60 times per second (50 times per second on most European 64s and 128s). The interrupt routine is called, and various housekeeping chores such as checking the keyboard and updating the jiffy clock are then performed. There are several other sources of interrupts as well.

In some cases, it's necessary to disable interrupts to forestall the possibility that an IRQ will happen. This is especially important in situations where a wedge is being installed or when character ROM is being read. The SEI instruction sets the interrupt flag to disable IRQs. CLI turns interrupts back on.

Note that the state of the I flag does not affect nonmaskable interrupts (NMIs).

CLV

CLear oVerflow: Clear the overflow flag.

Addressing Modes

Implied	CLV	B8	2 cycles

Flags

N (Negative)	—
V (Overflow)	Set to zero.
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

CLV clears the overflow flag (V) to zero, typically before an

operation involving signed arithmetic. Signed arithmetic handles numbers from −128 through 127. The negative numbers are $80–$FF (128 through 255 or −128 through −1); the positive numbers are $00–$7F (0–127).

When a number changes sign in signed arithmetic, an overflow occurs from bit 6 to bit 7 in the result, setting V. Frequently, at this point—perhaps after a BVS—a CLV is used to clear the flag.

CLV is sometimes used along with BVC to carry out a "branch always" (such as CLV: BVC).

CMP

CoMPare: Compare the number in .A with a value.

Addressing Modes

(Zero page,X)	CMP ($6B,X)	C1 6B	6 cycles
Zero page	CMP $55	C5 55	3 cycles
Immediate	CMP #$30	C9 30	2 cycles
Absolute	CMP $1CA8	CD A8 1C	4 cycles
(Zero page),Y	CMP ($F1),Y	D1 F1	5 cycles (+1 over a page)
Zero page,X	CMP $10,X	D5 10	4 cycles
Absolute,Y	CMP $1EFC,Y	D9 FC 1E	4 cycles (+1 over a page)
Absolute,X	CMP $9500,X	DD 00 95	4 cycles (+1 over a page)

Flags

N (Negative)	If .A minus the value is $80–$FF (or −128 through −1), N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .A equals the value, Z is set.
C (Carry)	If .A is greater than or equal to the value, C is set.

CMP compares the accumulator value with another number by subtracting the value from .A. The two values are not changed, and the result is thrown away. The operation does set three flags, however.

A very common use of CMP is to look for a specific value—CMP #$30: BEQ, for example. If .A holds a $30, the result of subtracting $30 is zero, and the Z flag will be set. The

BEQ then branches on if equal to zero. If the two numbers are not equal, the branch will not occur.

Another way to use CMP is to look for numbers within a certain range. If the number in .A is greater than or equal to the number being compared, the carry flag will be set. (See SBC for a discussion of how the C flag is used in subtraction.) If .A is less than the value, the C flag will be cleared. You can then use BCS or BCC to branch to the appropriate location.

CPX

ComPare .X: Compare .X with a value.

Addressing Modes

Immediate	CPX #$A9	E0 A9	2 cycles
Zero page	CPX $1F	E4 1F	3 cycles
Absolute	CPX $3002	EC 02 30	4 cycles

Flags

N (Negative)	If .X minus the value is $80–$FF, N is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .X equals the value, Z is set.
C (Carry)	If .X is greater than or equal to the value, C is set.

CPX subtracts the value from .X, discarding the result. In the process, three flags are set, based on the result of the subtraction. In most cases, CPX is used along with a branch instruction operating on the N, Z, or C flag.

CPY

ComPare .Y: Compare .Y with a value.

Addressing Modes

Immediate	CPY #$16	C0 16	2 cycles
Zero page	CPY $F0	C4 F0	3 cycles
Absolute	CPY $C020	CC 20 C0	4 cycles

Flags

N (Negative)	If .Y minus the value is $80–$FF, N is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—

I (Interrupt)	—
Z (Zero)	If .Y equals the value, Z is set.
C (Carry)	If .Y is greater than or equal to the value, C is set.

CPY performs the operation *.Y minus value,* without storing the result anywhere. The N, Z, and C flags are based on the result of the subtraction. CPY is most often used in conjunction with a branch instruction, especially in loops.

DEC

DECrement: Subtract one from a value.

Addressing Modes

Zero page	DEC $14	C6 14	5 cycles
Absolute	DEC $4707	CE 07 47	6 cycles
Zero page,X	DEC $30,X	D6 30	6 cycles
Absolute,X	DEC $5F02,X	DE 02 5F	7 cycles

Flags

N (Negative)	If the result is negative ($80–$FF), N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If the value holds a $01 and it counts to $00, Z is set.
C (Carry)	—

DEC decrements the contents of the specified byte by one, setting the N and Z flags based on the result. After counting down to zero, the next DEC yields a 255 (a negative number). For this reason, DEC is almost always used in loops which count down to zero (Z is set) or to one past zero (N is set).

DEX

DEcrement .X: Subtract one from the value in the .X register.

Addressing Modes

Implied	DEX	CA	2 cycles

Flags

N (Negative)	If the result is negative ($80–$FF), N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—

I (Interrupt)	—
Z (Zero)	If .X holds a $01 and it counts to $00, Z is set.
C (Carry)	—

DEX is used most often within loops that count from a given value down to zero or one past zero (255). If .X holds a zero, DEX causes it to wrap around to 255.

DEY

DEcrement .Y: Subtract one from the value in the .Y register.

Addressing Modes

Implied	DEY	88	2 cycles

Flags

N (Negative)	If the result is negative ($80–$FF), N is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .Y holds a $01 and it counts to $00, Z is set.
C (Carry)	—

In its application, DEY is similar to DEX. Like DEX, it's frequently found in counting loops that decrement to zero or to one past zero.

EOR

Exclusive OR: Perform a bitwise EOR between the accumulator and a value. The result is stored in the accumulator.

Addressing Modes

(Zero page,X)	EOR ($EB,X)	41	EB		6 cycles
Zero page	EOR $E9	45	E9		3 cycles
Immediate	EOR #$93	49	93		2 cycles
Absolute	EOR $8DA2	4D	A2	8D	4 cycles
(Zero page),Y	EOR ($C2),Y	51	C2		5 cycles (+1 over a page)
Zero page,X	EOR $2B,X	55	2B		4 cycles
Absolute,Y	EOR $CF88,Y	59	[redacted]	CF	4 cycles (+1 over a page)
Absolute,X	EOR $53E8,X	5D	E8	53	4 cycles (+1 over a page)

Flags

N (Negative)	If the result is \$80–\$FF, N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If the result is zero, Z is set.
C (Carry)	—

EOR is a bitwise operation like AND and ORA. It compares the bits in the accumulator with a value from memory and sets the resulting bits according to the logic of exclusive OR, which is *one or the other, but not both*. A one and a zero result in a bit that's set. But if both are zeros or both are ones, the result is a zero:

\$6E 0110 1110
\$16 0001 0110
\$78 0111 1000

In the example note that where bits are set in \$16 (bits 1, 2, and 4), the corresponding bits in \$6E are flipped. If you EOR a given bit with zero, the result is no change. But if you EOR with one, a zero becomes a one, and a one becomes a zero.

EOR's primary uses are in flipping specific bits of a memory location or register, and in encryption. If you EOR with a specific number and then EOR with the same number, you get back the original value. This property makes EOR valuable for encoding and decoding.

INC

INCrement: Add one to a value.

Addressing Modes

Zero page	INC \$2F	E6 2F	5 cycles
Absolute	INC \$BC0B	EE 0B BC	6 cycles
Zero page,X	INC \$24,X	F6 24	6 cycles
Absolute,X	INC \$BFFF,X	FE FF BF	7 cycles

Flags

N (Negative)	If the result is negative (\$80–\$FF), N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—

Z (Zero)	If the value holds an \$FF and it counts to \$00, Z is set.
C (Carry)	—

INC adds one to a memory location, almost invariably a counter byte. If the byte holds a 255 (\$FF), it wraps around to zero. This makes it ideal for loops where the .X and .Y registers are already being used (thus precluding use of INX and INY).

INX

INcrement .X: Add one to the value in .X.

Addressing Modes

Implied	INX	E8	2 cycles

Flags

N (Negative)	If the result is between \$80 and \$FF, the N flag is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .X counts from \$FF through \$00, the Z flag is set.
C (Carry)	—

INX adds one to the value in the .X register. If .X currently holds a 255 (\$FF), the value wraps around to zero. INX is usually found inside loops that count forward, where .X may be involved in an indexed load or store.

INY

INcrement .Y: Add one to the value in .Y.

Addressing Modes

Implied	INY	C8	2 cycles

Flags

N (Negative)	If the result is \$80–\$FF, the N flag is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .Y counts from \$FF to \$00, the Z flag is set.
C (Carry)	—

INY adds one to the Y register, causing it to turn over to zero when 255 ($FF) is reached. As with INX, this makes it ideal for loops branching on the N or Z flag.

JMP

JuMP: Jump to a given address.

Addressing Modes

Absolute	JMP $6299	4C 99 62	3 cycles
(Absolute)	JMP ($0E08)	6C 08 0E	5 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

JMP changes the value in the program counter; the next instruction to be executed will come from the address provided. JMP is the ML equivalent of BASIC's GOTO.

An absolute jump just moves to the address indicated. An indirect jump—JMP ($060C), for example—loads the two-byte address from the given *vector* and jumps there. If $060C contains a $D2 and $060D has an $FF, the indirect jump will combine the low byte and the high byte and go to $FFD2.

Because of a bug in the 6502, you should avoid putting indirect jumps directly into a program that assembles to unknown memory locations. If the vector falls on a page boundary (say, $08FF–$0900), the low byte will be loaded from $08FF as expected, but the high byte will come from $0800, not from $0900. In a case like this, there's no telling where the indirect jump will go. The best policy is to put vectors at known addresses.

Many 64 and 128 routines use indirect jump vectors in RAM. Most are found in page 3 ($0300–$03FF).

JSR

Jump to SubRoutine: Jump to a given address, saving the return address.

Addressing Modes

Absolute	JSR $6E01	20 01 6E	6 cycles

Flags

N (Negative) —
V (Overflow) —
\- —
B (Break) —
D (Decimal) —
I (Interrupt) —
Z (Zero) —
C (Carry) —

JSR changes the program counter to the address specified. A return address, pointing to the instruction following the JSR, is left on the stack. GOSUB is the BASIC equivalent of JSR.

JSR is used primarily when a section of code is used repeatedly in a program. Rather than the code being replicated each time it's needed, it's set apart from the main program as a subroutine, typically ending with RTS and called with JSR.

To speed up your program a little and save a byte of memory, you may replace any JSR followed directly by an RTS with a JMP instruction. For example, instead of JSR $FFD2: RTS, you may use JMP $FFD2—in effect, borrowing the RTS at the end of the $FFD2 routine.

LDA

LoaD the Accumulator: Put a value into .A.

Addressing Modes

(Zero page,X)	LDA ($7B,X)	**A1** 7B	6 cycles
Zero page	LDA $77	**A5** 77	3 cycles
Immediate	LDA #$02	**A9** 02	2 cycles
Absolute	LDA $DBC2	**AD** C2 DB	4 cycles
(Zero page),Y	LDA ($DF),Y	**B1** DF	5 cycles (+1 over a page)
Zero page,X	LDA $6D,X	**B5** 6D	4 cycles
Absolute,Y	LDA $0AEF,Y	**B9** EF 0A	4 cycles (+1 over a page)
Absolute,X	LDA $3D77	**BD** 77 3D	4 cycles (+1 over a page)

Flags

N (Negative) If the value is negative ($80–$FF), N is set.
V (Overflow) —
\- —
B (Break) —
D (Decimal) —
I (Interrupt) —

Z (Zero)	If the value is a zero, Z is set.
C (Carry)	—

LDA is one of the most widely used instructions. It loads a number from memory into the accumulator. (Immediate mode loads a specified number into .A; in this case, the number is part of the program, following immediately after the $A9 opcode.)

Usually, the value loaded is soon stored into memory with STA, although it may also be used in a math operation like ADC, AND, EOR, ORA, SBC, or the like.

LDX

LoaD .X: Load a value into the X register.

Addressing Modes

Immediate	LDX #$BB	A2 BB	2 cycles
Zero page	LDX $7A	A6 7A	3 cycles
Absolute	LDX $A808	AE 08 A8	4 cycles
(Zero page),Y	LDX ($FD),Y	B6 FD	4 cycles
Absolute,Y	LDX $3F09,Y	BE 09 3F	4 cycles (+1 over a page)

Flags

N (Negative)	If the value is $80–$FF, N is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .X is loaded with a zero, Z is set.
C (Carry)	—

LDX loads a specific value into the X register. Common uses are in transferring data from temporary locations or onto the stack (LDX: TXS), in initializing counter loops, or in setting up an offset for indexed addressing.

LDY

LoaD .Y: Load a value into the Y register.

Addressing Modes

Immediate	LDY #$A5	A0 A5	2 cycles
Zero page	LDY $12	A4 12	3 cycles
Absolute	LDY $0BF5	AC F5 0B	4 cycles
Zero page,X	LDY $39,X	B4 39	4 cycles
Absolute,X	LDY $133B,X	BC 3B 13	4 cycles (+1 over a page)

Flags

N (Negative)	If the value is \$80–\$FF, N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .Y is loaded with a zero, Z is set.
C (Carry)	—

The LDY instruction puts a given number into the Y register. Most often, you'll see immediate addressing in preparation for a loop indexed by .Y. Either .Y is loaded with zero (for a loop that counts forward with INY) or with a specific number (for a loop that counts down with DEY).

LSR

Logical Shift Right: Shift a value (accumulator or memory) to the right.

Addressing Modes

Zero page	LSR \$A3	**46**	A3		5 cycles
Accumulator	LSR	**4A**			2 cycles
Absolute	LSR \$CA06	**4E**	06	CA	6 cycles
Zero page,X	LSR \$DD,X	**56**	DD		6 cycles
Absolute,X	LSR \$5D02,X	**5E**	02	5D	7 cycles

Flags

N (Negative)	Set to zero.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If the value is \$01 or \$00, Z is set.
C (Carry)	Bit 0 shifts into carry and sets/clears the C flag.

The LSR instruction shifts all eight bits one position to the right, placing a zero in bit 7 and moving bit 0 into the carry flag.

A frequent application of LSR is to test bit 0 and branch accordingly (LSR: BCS/BCC). But LSR probably finds its greatest use in certain mathematical manipulations: converting negative numbers to positive (LSR: ROL), dividing bytes by 2 with the remainder placed in C, and shifting the high nybble of a byte into the low nybble (LSR: LSR: LSR: LSR).

NOP

No OPeration: Do nothing.

Addressing Modes

Implied	NOP	EA	2 cycles

Flags

N (Negative)	—
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

After a NOP, the values in memory, the numbers in the registers, and the status flags remain the same. The program counter advances by one. NOP is sometimes used to remove part of a program. If three bytes hold a JSR instruction, you can POKE NOPs on top of the memory there, and the program will not execute the JSR. NOPs are also found in delay loops where the timing is finely tuned.

ORA

Bitwise OR: Perform a bitwise OR between .A and a value, storing the result in .A.

Addressing Modes

(Zero page,X)	ORA ($1B,X)	01	1B		6 cycles
Zero page	ORA $68	05	68		3 cycles
Immediate	ORA #$3F	09	3F		2 cycles
Absolute	ORA $BA03	0D	03	BA	4 cycles
(Zero page),Y	ORA ($4C),Y	11	4C		5 cycles
Zero page,X	ORA $63,X	15	63		4 cycles
Absolute,Y	ORA $4E0F,Y	19	0F	4E	4 cycles (+1 over a page)
Absolute,X	ORA $2A0B,X	1D	0B	2A	4 cycles (+1 over a page)

Flags

N (Negative)	If bit 7 is set, the N flag is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—

Z (Zero) If the result is zero, Z is set.
C (Carry) —

ORA performs a bitwise OR on a value. Corresponding bits in .A and the value are compared. If either bit is on, the result is one.

For instance, to turn on bits 0 and 1 in $BC, you would ORA with $03:

$BC 1011 1100
$03 0000 0011
$BF 1011 1111

To turn certain bits off, use AND.

PHA

PusH .A: Push the current value of the accumulator onto the stack. The accumulator is not changed. The stack pointer decreases by one.

Addressing Modes

Implied	PHA	48	3 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

PHA pushes .A onto the stack. No flags are affected. A common use for PHA is to temporarily save the number in the accumulator. You push it, do something else, then pull it back. Another, more sophisticated technique is to push two values onto the stack and then execute an RTS. RTS returns from a subroutine to the original program that called the subroutine. It does so by pulling the program counter (minus one) from the stack. If PHA has put a valid address on the stack, RTS will return to the address you have provided. Push the high byte first, then the low byte of the address (minus one) of the routine you wish to call.

PHP

PusH Processor status register: Push the value in the processor's status register onto the stack. The stack pointer decreases by one.

Addressing Modes

Implied	PHP	08	3 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

PHP stores the contents of the status register on the stack, affecting no flags. The processor status register (P) contains all the flags (N, V, B, D, I, Z, and C).

PHP is the complementary instruction to PLP, which pulls a stack byte into the status register. When status bits are being tested, PHP and PLP are often found in tandem, especially when intervening instructions are apt to affect these bits.

For instance, suppose you wished to branch, based on the N flag following a particular instruction, but operations that affect the status flag are necessary prior to the branch. To preserve the status register for later testing, you would push it onto the stack with PHP, proceed with the interfering operations, and then restore it with PLP just before the branch.

When using this approach, remember not to use other stack-oriented instructions like JSR, RTS, or RTI before the PLP has executed.

PLA

PuLl .A: Pull a value from the stack into the accumulator.

Addressing Modes

Implied	PLA	68	4 cycles

Flags

N (Negative)	If the number is negative, N is set to one.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—

I (Interrupt)	—
Z (Zero)	If a zero is pulled, Z is set.
C (Carry)	—

PLA pulls values off the stack. It is the opposite of PHA, which pushes numbers there. After the PLA, the stack pointer is increased by one.

PHA and PLA are useful for temporarily storing the current status of the accumulator. You push a value onto the stack, perform some other operation, and then pull it back into .A. However, you should be careful that you don't perform other stack-oriented operations such as JSR, RTS, or RTI, in the meantime.

PHA and PLA can also be used to set up and destroy addresses for RTS. You may JSR to a routine only to find that (in special cases) it's not necessary to RTS back to the calling routine. Two PLAs will remove the return address from the stack. (JSR pushes the return address minus one onto the stack, high byte first, and RTS pulls the two bytes.)

PLP

PuLl Processor status register: Pull a value from the stack into the processor's status register.

Addressing Modes

Implied	PLP	**28**	4 cycles

Flags

N (Negative)	If the number is negative, N is set.
V (Overflow)	If bit 6 is on, V is set.
–	—
B (Break)	If bit 4 is on, B is set.
D (Decimal)	If bit 3 is on, D is set.
I (Interrupt)	If bit 2 is on, I is set.
Z (Zero)	If bit 1 is on, Z is set.
C (Carry)	If bit 0 is on, C is set.

PLP takes a byte from the stack, placing it in the status register. The stack pointer increments by one.

PLP is the opposite of PHP, which pushes the contents of the status register onto the stack. These two are frequently used together, much like PHA/PLA.

PLP's role in this arrangement is to retrieve the status register after it has been pushed onto the stack with PHP. Typically in this situation a branching instruction will follow.

ROL

ROtate Left: Rotate a value (accumulator or memory) to the left.

Addressing Modes

Zero page	ROL $3A	26 3A	5 cycles
Accumulator	ROL	2A	2 cycles
Absolute	ROL $8FA6	2E A6 8F	6 cycles
Zero page,X	ROL $46,X	36 46	6 cycles
Absolute,X	ROL $0EFB,X	3E FB 0E	7 cycles

Flags

N (Negative)	Bit 6 rotates into 7 and sets/clears the N flag.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If carry is clear and bits 0–6 are zero, Z is set.
C (Carry)	Bit 7 rotates into carry.

ROL causes all eight bits to rotate one position to the left. The carry flag moves into bit 0, and bit 7 moves into the carry flag. ROL is most commonly used in two-byte shifts: You ASL the low byte and ROL the high byte.

ROR

ROtate Right: Rotate a value (accumulator or memory) to the right.

Addressing Modes

Zero page	ROR $13	66 13	5 cycles
Accumulator	ROR	6A	2 cycles
Absolute	ROR $BB67	6E 67 BB	6 cycles
Zero page,X	ROR $E1,X	76 E1	6 cycles
Absolute,X	ROR $1110,X	7E 10 11	7 cycles

Flags

N (Negative)	Carry rotates into bit 7 and sets/clears the N flag.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If carry is clear and bits 1–7 are zero, Z is set.
C (Carry)	Bit 0 rotates into carry.

ROR is the complement instruction to ROL: It shifts all eight bits one position to the right. Bit 0 moves into the carry flag, and carry shifts into bit 7.

ROR is used to carry out two-byte shifts (to halve a number). You first LSR the high byte and then ROR the low byte. Also, ROR often precedes testing of the N, Z, or C flag.

RTI

ReTurn from Interrupt: Restore the processor status and the program counter.

Addressing Modes

Implied	RTI	40	6 cycles

Flags

N (Negative)	Reset to its status before the interrupt.
V (Overflow)	Reset to its status before the interrupt.
–	—
B (Break)	Reset to its status before the interrupt.
D (Decimal)	Reset to its status before the interrupt.
I (Interrupt)	Reset to its status before the interrupt.
Z (Zero)	Reset to its status before the interrupt.
C (Carry)	Reset to its status before the interrupt.

When an interrupt occurs, the current program counter (high byte, then low byte) is pushed onto the stack, followed by the processor status (P), where all the flags are located.

RTI causes .P to be pulled from the stack, followed by the program counter. The program then continues at one byte beyond the address pulled from the stack.

RTS

ReTurn from Subroutine: Reset the program counter using the return address on the stack.

Addressing Modes

Implied	RTS	60	6 cycles

Flags

N (Negative)	—
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—

Z (Zero) —
C (Carry) —

RTS removes the last two bytes from the stack (low byte first, then high byte), adds 1 to the resulting address, and places it in the program counter. The stack pointer increments by 2, and program execution continues at the return address in the program counter. Unlike RTI, the RTS instruction affects no flags.

RTS is used almost exclusively to return from a subroutine, whether called from within the ML with JSR or from BASIC with SYS. When an ML subroutine is called from BASIC, the return address for BASIC's main loop is first placed on the stack. So, once the ML routine is complete, a return to the BASIC program successfully occurs.

Another application of RTS involves simulating a JMP instruction. With PHA, you push the high bytes and low bytes of a routine you wish to jump to onto the stack. (Because RTS adds 1 to the address it finds, you *must* subtract 1 from the actual address of the routine you're calling before pushing the address onto the stack.) When the next RTS executes, the program continues, using the address on the stack. Take care that you don't put extra bytes on the stack before the RTS.

SBC

SuBtract with Carry: Subtract a value from the accumulator, with the result in .A.

Addressing Modes

(Zero page,X)	SBC ($8A,X)	E1 8A	6 cycles
Zero page	SBC $1A	E5 1A	3 cycles
Immediate	SBC #$B7	E9 B7	2 cycles
Absolute	SBC $6862	ED 62 68	4 cycles
(Zero page),Y	SBC ($E1),Y	F1 E1	5 cycles (+1 over a page)
Zero page,X	SBC ($D6),X	F5 D6	4 cycles
Absolute,Y	SBC $80EB,Y	F9 EB 80	4 cycles (+1 over a page)
Absolute,X	SBC $7088	FD 88 70	4 cycles (+1 over a page)

Flags

N (Negative) If the result is $80–$FF, the N flag is set.
V (Overflow) If an overflow occurs, V is set.
- —
B (Break) —
D (Decimal) —

I (Interrupt) —
Z (Zero) If the result is zero, Z is set.
C (Carry) If .A is greater than or equal to the value subtracted, the result is positive, and C is set.

The rule to remember is always to clear the carry flag (CLC) before addition and always to set the carry flag (SEC) before subtraction. If you're subtracting large numbers (two bytes or more), set carry before subtracting the least significant byte. As larger numbers are subtracted, carry will take care of itself.

Subtracting a large number from a smaller number (5 − 20, for example) will result in a cleared carry. If the second number is smaller than the first, carry will remain set.

The result of the subtraction is found in the accumulator; if you want to save the number, be sure to STA after the subtraction.

SEC

SEt Carry: Set the carry flag.

Addressing Modes

Implied	SEC	38	2 cycles

Flags

N (Negative) —
V (Overflow) —
- —
B (Break) —
D (Decimal) —
I (Interrupt) —
Z (Zero) —
C (Carry) Set to one.

SEC, the complementary instruction to CLC, sets the carry flag. This is necessary in order for SBC to work correctly (for a "borrow"). SEC can also force a branch (SEC: BCS), or it may be used along with the rotate instructions (ROL, ROR). Additionally, some Kernal routines set carry with SEC to indicate that an error has occurred.

SED

SEt Decimal mode: Turns on binary-coded decimal (BCD) mode.

Addressing Modes

Implied	SED	F8	2 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	Set to one.
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

SED turns on BCD mode, where bytes are allowed to have 100 values ($00–$99) instead of 255 ($00–$FF). When the decimal flag is turned on, addition and subtraction act only on the numbers 0–9. If you add 1 to $09 in decimal mode, the result is $10, not $0A. Individual nybbles are allowed to hold the numbers $0–$9 instead of $0–$F.

To turn off the D flag, use CLD.

SEI

SEt Interrupt flag: Disable maskable (IRQ) interrupts.

Addressing Modes

Implied	SEI	78	2 cycles

Flags

N (Negative)	—
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	Set to one.
Z (Zero)	—
C (Carry)	—

Every 1/60 second (or 1/50 second on most European 64s and 128s), an interrupt request (IRQ) occurs. At this time, a service routine handles various housekeeping chores like updating the jiffy clock and the screen, or checking the keyboard.

SEI prevents the normal IRQ interrupts from being honored by setting the I flag. (Nonmaskable interrupts—NMIs—like BRK are still active.) Frequently, it is necessary to set this flag before certain vectors are changed.

Turn interrupts back on with CLI.

STA

STore Accumulator: Copy the contents of .A to memory.

Addressing Modes

(Zero page,X)	STA ($F6,X)	81	F6		6 cycles
Zero page	STA $2D	85	2D		3 cycles
Absolute	STA $B8F6	8D	F6	B8	4 cycles
(Zero page),Y	STA ($DF),Y	91	DF		6 cycles
Zero page,X	STA $4E,X	95	4E		4 cycles
Absolute,Y	STA $3EA5,Y	99	A5	3E	5 cycles
Absolute,X	STA $7534,X	9D	34	75	5 cycles

Flags

N	(Negative)	—
V	(Overflow)	—
-		—
B	(Break)	—
D	(Decimal)	—
I	(Interrupt)	—
Z	(Zero)	—
C	(Carry)	—

STA and LDA are probably the two most common instructions in ML. LDA puts a value into the accumulator; STA stores the value from .A into memory. The contents of the accumulator remain unchanged after the store.

STX

STore .X: Store the value in the X register to memory.

Addressing Modes

Zero page	STX $C6	86	C6		3 cycles
Absolute	STX $6D0E	8E	0E	6D	4 cycles
Zero page,Y	STX $FA,Y	96	FA		4 cycles

Flags

N	(Negative)	—
V	(Overflow)	—
-		—
B	(Break)	—
D	(Decimal)	—
I	(Interrupt)	—
Z	(Zero)	—
C	(Carry)	—

STX puts the value currently in .X into memory. No flags or data registers are affected. STX is similar in its applications to

STA, temporarily storing the contents of the register to memory or initializing memory to a set value. Note that STX has far fewer addressing modes than does STA. Because loading and storing from .A is more flexible, the X register is most often used as a counter or as an index.

STY

STore .Y: Store the value in the Y register to memory.

Addressing Modes

Zero page	STY $9E	84 9E	3 cycles
Absolute	STY $6F17	8C 17 6F	4 cycles
Zero page,X	STY $58,X	94 58	4 cycles

Flags

N (Negative)	—
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

STY takes the value in .Y and stores it to memory. The Y register is not affected. STY is sometimes helpful when the index value needs to be saved (before a subroutine that changes the registers), but it really isn't used very often.

TAX

Transfer .A to .X: Copy the value in the accumulator to the X register.

Addressing Modes

Implied	TAX	AA	2 cycles

Flags

N (Negative)	If .A holds $80–$FF, N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .A holds a zero, Z is set.
C (Carry)	—

TAX moves the value in .A to .X. This instruction is handy for temporarily storing the contents of the accumulator or for initializing .X when indexed addressing is used.

TAY

Transfer .A to .Y: Moves the value in the accumulator to .Y.

Addressing Modes

Implied	TAY	**A8**	2 cycles

Flags

N (Negative)	If .A is negative ($80–$FF), the N flag is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .A is zero, this flag is set.
C (Carry)	—

TAY copies the value in .A to .Y. The original value in the accumulator remains unchanged. Some programmers use this technique to temporarily save the value of .A. Another use is to set up an indexed LDA from a table.

TSX

Transfer Stack pointer to .X: Copy the value in the stack pointer to the X register.

Addressing Modes

Implied	TSX	**BA**	2 cycles

Flags

N (Negative)	If the stack pointer is $80–$FF, the N flag is set.
V (Overflow)	—
-	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If the stack pointer is zero, Z is set.
C (Carry)	—

TSX moves the stack pointer into .X. The stack pointer itself is a single byte, offset to $0100.

One application of TSX is to determine the amount of space remaining on the stack. Another is to examine the contents of the stack. (Use TSX: LDA $0100,X to look at the last

value placed on the stack.) Still a third application involves saving the current stack pointer while using a portion of the stack for certain operations.

TXA

Transfer .X to .A: Moves the value in .X to the accumulator, leaving .X unchanged.

Addressing Modes

Implied	TXA	**8A**	2 cycles

Flags

N (Negative)	If the value transferred is \$80–\$FF, N is set.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .X holds a 00, the Z flag is set.
C (Carry)	—

TXA moves the number currently in .X to .A. The value in .X remains the same. This is sometimes done in preparation for an instruction such as ADC, PHA, SBC, or some other operation that cannot be performed directly on the .X register.

TXS

Transfer .X to Stack pointer: Copy the value in the .X register to the stack pointer.

Addressing Modes

Implied	TXS	**9A**	2 cycles

Flags

N (Negative)	—
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	—
C (Carry)	—

TXS moves the contents of the .X register into the stack pointer. This instruction is used by the computer as part of its own power-up routine. The stack pointer is set to the top of the stack (which is called *clearing the stack*) when the com-

puter is first turned on or RESET with LDX #$FF: TXS.

TXS is also helpful in restoring the stack pointer after any processing has been carried out within the stack itself.

TYA

Transfer .Y to .A: Copy the value in the Y register to the accumulator; .Y remains unchanged.

Addressing Modes

Implied	TYA	98	2 cycles

Flags

N (Negative)	If .Y holds $80–$FF, N is set. If .Y is $00–$7F, N is clear.
V (Overflow)	—
–	—
B (Break)	—
D (Decimal)	—
I (Interrupt)	—
Z (Zero)	If .Y holds a zero, Z is set.
C (Carry)	—

TYA moves the value in .Y to .A. This is sometimes necessary because the accumulator can perform some operations (like addition and subtraction) that aren't available for .Y.

Opcodes Listed Numerically

Opcode			Mnemonic	Addressing Mode
00			BRK	Implied
01	ZX		ORA	(Zero page,X)
02			Undefined	—
03			Undefined	—
04			Undefined	—
05	ZP		ORA	Zero page
06	ZP		ASL	Zero page
07			Undefined	—
08			PHP	Implied
09	IM		ORA	Immediate
0A			ASL	Accumulator
0B			Undefined	—
0C			Undefined	—
0D	LO	HI	ORA	Absolute
0E	LO	HI	ASL	Absolute
0F			Undefined	—
10	RE		BPL	Relative
11	ZY		ORA	(Zero page),Y
12			Undefined	—
13			Undefined	—
14			Undefined	—
15	ZP		ORA	Zero page,X
16	ZP		ASL	Zero page,X
17			Undefined	—
18			CLC	Implied
19	LO	HI	ORA	Absolute,Y
1A			Undefined	—
1B			Undefined	—
1C			Undefined	—
1D	LO	HI	ORA	Absolute,X
1E	LO	HI	ASL	Absolute,X
1F			Undefined	—
20	LO	HI	JSR	Absolute
21	ZX		AND	(Zero page,X)
22			Undefined	—
23			Undefined	—
24	ZP		BIT	Zero page
25	ZP		AND	Zero page
26	ZP		ROL	Zero page
27			Undefined	—
28			PLP	Implied
29	IM		AND	Immediate
2A			ROL	Accumulator
2B			Undefined	—

Opcode	Mnemonic	Addressing Mode
2C LO HI	BIT	Absolute
2D LO HI	AND	Absolute
2E LO HI	ROL	Absolute
2F	Undefined	—
30 RE	BMI	Relative
31 ZY	AND	(Zero page),Y
32	Undefined	—
33	Undefined	
34	Undefined	—
35 ZP	AND	Zero page,X
36 ZP	ROL	Zero page,X
37	Undefined	—
38	SEC	Implied
39 LO HI	AND	Absolute,Y
3A	Undefined	—
3B	Undefined	—
3C	Undefined	—
3D LO HI	AND	Absolute,X
3E LO HI	ROL	Absolute,X
3F	Undefined	—
40	RTI	Implied
41 ZX	EOR	(Zero page,X)
42	Undefined	—
43	Undefined	—
44	Undefined	—
45 ZP	EOR	Zero page
46 ZP	LSR	Zero page
47	Undefined	—
48	PHA	Implied
49 IM	EOR	Immediate
4A	LSR	Accumulator
4B	Undefined	—
4C LO HI	JMP	Absolute
4D LO HI	EOR	Absolute
4E LO HI	LSR	Absolute
4F	Undefined	—
50 RE	BVC	Relative
51 ZY	EOR	(Zero page),Y
52	Undefined	—
53	Undefined	—
54	Undefined	—
55 ZP	EOR	Zero page,X
56 ZP	LSR	Zero page,X
57	Undefined	—
58	CLI	Implied

Opcode			Mnemonic	Addressing Mode
59	LO	HI	EOR	Absolute,Y
5A			Undefined	—
5B			Undefined	—
5C			Undefined	—
5D	LO	HI	EOR	Absolute,X
5E	LO	HI	LSR	Absolute,X
5F			Undefined	—
60			RTS	Implied
61	ZX		ADC	(Zero page,X)
62			Undefined	—
63			Undefined	—
64			Undefined	—
65	ZP		ADC	Zero page
66	ZP		ROR	Zero page
67			Undefined	—
68			PLA	Implied
69	IM		ADC	Immediate
6A			ROR	Accumulator
6B			Undefined	—
6C	LO	HI	JMP	(Absolute)
6D	LO	HI	ADC	Absolute
6E	LO	HI	ROR	Absolute
6F			Undefined	—
70	RE		BVS	Relative
71	ZY		ADC	(Zero page),Y
72			Undefined	—
73			Undefined	—
74			Undefined	—
75	ZP		ADC	Zero page,X
76	ZP		ROR	Zero page,X
77			Undefined	—
78			SEI	Implied
79	LO	HI	ADC	Absolute,Y
7A			Undefined	—
7B			Undefined	—
7C			Undefined	—
7D	LO	HI	ADC	Absolute,X
7E	LO	HI	ROR	Absolute,X
7F			Undefined	—
80			Undefined	—
81	ZX		STA	(Zero page,X)
82			Undefined	—
83			Undefined	—
84	ZP		STY	Zero page
85	ZP		STA	Zero page

Opcode			Mnemonic	Addressing Mode
86	ZP		STX	Zero page
87			Undefined	—
88			DEY	Implied
89			Undefined	—
8A			TXA	Implied
8B			Undefined	—
8C	LO	HI	STY	Absolute
8D	LO	HI	STA	Absolute
8E	LO	HI	STX	Absolute
8F			Undefined	—
90	RE		BCC	Relative
91	ZY		STA	(Zero page),Y
92			Undefined	—
93			Undefined	—
94	ZP		STY	Zero page,X
95	ZP		STA	Zero page,X
96	ZP		STX	Zero page,Y
97			Undefined	—
98			TYA	Implied
99	LO	HI	STA	Absolute,Y
9A			TXS	Implied
9B			Undefined	—
9C			Undefined	—
9D	LO	HI	STA	Absolute,X
9E			Undefined	—
9F			Undefined	—
A0	IM		LDY	Immediate
A1	ZX		LDA	(Zero page,X)
A2	IM		LDX	Immediate
A3			Undefined	—
A4	ZP		LDY	Zero page
A5	ZP		LDA	Zero page
A6	ZP		LDX	Zero page
A7			Undefined	—
A8			TAY	Implied
A9	IM		LDA	Immediate
AA			TAX	Implied
AB			Undefined	—
AC	LO	HI	LDY	Absolute
AD	LO	HI	LDA	Absolute
AE	LO	HI	LDX	Absolute
AF			Undefined	—
B0	RE		BCS	Relative
B1	ZY		LDA	(Zero page),Y
B2			Undefined	—

Opcode	Mnemonic	Addressing Mode
B3	Undefined	—
B4 ZP	LDY	Zero page,X
B5 ZP	LDA	Zero page,X
B6 ZP	LDX	Zero page,Y
B7	Undefined	—
B8	CLV	Implied
B9 LO HI	LDA	Absolute,Y
BA	TSX	Implied
BB	Undefined	—
BC LO HI	LDY	Absolute,X
BD LO HI	LDA	Absolute,X
BE LO HI	LDX	Absolute,Y
BF	Undefined	—
C0 IM	CPY	Immediate
C1 ZX	CMP	(Zero page,X)
C2	Undefined	—
C3	Undefined	—
C4 ZP	CPY	Zero page
C5 ZP	CMP	Zero page
C6 ZP	DEC	Zero page
C7	Undefined	—
C8	INY	Implied
C9 IM	CMP	Immediate
CA	DEX	Implied
CB	Undefined	—
CC LO HI	CPY	Absolute
CD LO HI	CMP	Absolute
CE LO HI	DEC	Absolute
CF	Undefined	—
D0 RE	BNE	Relative
D1 ZY	CMP	(Zero page),Y
D2	Undefined	—
D3	Undefined	—
D4	Undefined	—
D5 ZP	CMP	Zero page,X
D6 ZP	DEC	Zero page,X
D7	Undefined	—
D8	CLD	Implied
D9 LO HI	CMP	Absolute,Y
DA	Undefined	—
DB	Undefined	—
DC	Undefined	—
DD LO HI	CMP	Absolute,X
DE LO HI	DEC	Absolute,X
DF	Undefined	—

Opcode	Mnemonic	Addressing Mode
E0 IM	CPX	Immediate
E1 ZX	SBC	(Zero page,X)
E2	Undefined	—
E3	Undefined	—
E4 ZP	CPX	Zero page
E5 ZP	SBC	Zero page
E6 ZP	INC	Zero page
E7	Undefined	—
E8	INX	Implied
E9 IM	SBC	Immediate
EA	NOP	Implied
EB	Undefined	—
EC LO HI	CPX	Absolute
ED LO HI	SBC	Absolute
EE LO HI	INC	Absolute
EF	Undefined	—
F0 RE	BEQ	Relative
F1 ZY	SBC	(Zero page),Y
F2	Undefined	—
F3	Undefined	—
F4	Undefined	—
F5 ZP	SBC	Zero page,X
F6 ZP	INC	Zero page,X
F7	Undefined	—
F8	SED	Implied
F9 LO HI	SBC	Absolute,Y
FA	Undefined	—
FB	Undefined	—
FC	Undefined	—
FD LO HI	SBC	Absolute,X
FE LO HI	INC	Absolute,X
FF	Undefined	—

Instructions Arranged Alphabetically

Mnemonic	Addressing Mode	Opcode
ADC	Absolute	6D LO HI
ADC	Absolute,X	7D LO HI
ADC	Absolute,Y	79 LO HI
ADC	Immediate	69 IM
ADC	Zero page	65 ZP
ADC	Zero page,X	75 ZP
ADC	(Zero page,X)	61 ZX
ADC	(Zero page),Y	71 ZY
AND	Absolute	2D LO HI
AND	Absolute,X	3D LO HI
AND	Absolute,Y	39 LO HI
AND	Immediate	29 IM
AND	Zero page	25 ZP
AND	Zero page,X	35 ZP
AND	(Zero page,X)	21 ZX
AND	(Zero page),Y	31 ZY
ASL	Absolute	0E LO HI
ASL	Absolute,X	1E LO HI
ASL	Accumulator	0A
ASL	Zero page	06 ZP
ASL	Zero page,X	16 ZP
BCC	Relative	90 RE
BCS	Relative	B0 RE
BEQ	Relative	F0 RE
BIT	Absolute	2C LO HI
BIT	Zero page	24 ZP
BMI	Relative	30 RE
BNE	Relative	D0 RE
BPL	Relative	10 RE
BRK	Implied	00
BVC	Relative	50 RE
BVS	Relative	70 RE
CLC	Implied	18
CLD	Implied	D8
CLI	Implied	58
CLV	Implied	B8
CMP	Absolute	CD LO HI
CMP	Absolute,X	DD LO HI
CMP	Absolute,Y	D9 LO HI
CMP	Immediate	C9 IM
CMP	Zero page	C5 ZP
CMP	Zero page,X	D5 ZP
CMP	(Zero page,X)	C1 ZX
CMP	(Zero page),Y	D1 ZY

Mnemonic	Addressing Mode	Opcode
CPX	Absolute	EC LO HI
CPX	Immediate	E0 IM
CPX	Zero page	E4 ZP
CPY	Absolute	CC LO HI
CPY	Immediate	C0 IM
CPY	Zero page	C4 ZP
DEC	Absolute	CE LO HI
DEC	Absolute,X	DE LO HI
DEC	Zero page	C6 ZP
DEC	Zero page,X	D6 ZP
DEX	Implied	CA
DEY	Implied	88
EOR	Absolute	4D LO HI
EOR	Absolute,X	5D LO HI
EOR	Absolute,Y	59 LO HI
EOR	Immediate	49 IM
EOR	Zero page	45 ZP
EOR	Zero page,X	55 ZP
EOR	(Zero page,X)	41 ZX
EOR	(Zero page),Y	51 ZY
INC	Absolute	EE LO HI
INC	Absolute,X	FE LO HI
INC	Zero page	E6 ZP
INC	Zero page,X	F6 ZP
INX	Implied	E8
INY	Implied	C8
JMP	Absolute	4C LO HI
JMP	(Absolute)	6C LO HI
JSR	Absolute	20 LO HI
LDA	Absolute	AD LO HI
LDA	Absolute,X	BD LO HI
LDA	Absolute,Y	B9 LO HI
LDA	Immediate	A9 IM
LDA	Zero page	A5 ZP
LDA	Zero page,X	B5 ZP
LDA	(Zero page,X)	A1 ZX
LDA	(Zero page),Y	B1 ZY
LDX	Absolute	AE LO HI
LDX	Absolute,Y	BE LO HI
LDX	Immediate	A2 IM
LDX	Zero page	A6 ZP
LDX	Zero page,Y	B6 ZP
LDY	Absolute	AC LO HI
LDY	Absolute,X	BC LO HI
LDY	Immediate	A0 IM

Mnemonic	Addressing Mode	Opcode
LDY	Zero page	A4 ZP
LDY	Zero page,X	B4 ZP
LSR	Absolute	4E LO HI
LSR	Absolute,X	5E LO HI
LSR	Accumulator	4A
LSR	Zero page	46 ZP
LSR	Zero page,X	56 ZP
NOP	Implied	EA
ORA	Absolute	0D LO HI
ORA	Absolute,X	1D LO HI
ORA	Absolute,Y	19 LO HI
ORA	Immediate	09 IM
ORA	Zero page	05 ZP
ORA	Zero page,X	15 ZP
ORA	(Zero page,X)	11 ZY
ORA	(Zero page),Y	11 ZY
PHA	Implied	48
PHP	Implied	08
PLA	Implied	68
PLP	Implied	28
ROL	Absolute	2E LO HI
ROL	Absolute,X	3E LO HI
ROL	Accumulator	2A
ROL	Zero page	26 ZP
ROL	Zero page,X	36 ZP
ROR	Absolute	6E LO HI
ROR	Absolute,X	7E LO HI
ROR	Accumulator	6A
ROR	Zero page	66 ZP
ROR	Zero page,X	76 ZP
RTI	Implied	40
RTS	Implied	60
SBC	Absolute	ED LO HI
SBC	Absolute,X	FD LO HI
SBC	Absolute,Y	F9 LO HI
SBC	Immediate	E9 IM
SBC	Zero page	E5 ZP
SBC	Zero page,X	F5 ZP
SBC	(Zero page,X)	E1 ZX
SBC	(Zero page),Y	F1 ZY
SEC	Implied	38
SED	Implied	F8
SEI	Implied	78
STA	Absolute	8D LO HI
STA	Absolute,X	9D LO HI

Mnemonic	Addressing Mode	Opcode		
STA	Absolute,Y	99	LO	HI
STA	Zero page	85	ZP	
STA	Zero page,X	95	ZP	
STA	(Zero page,X)	81	ZX	
STA	(Zero page),Y	91	ZY	
STX	Absolute	8E	LO	HI
STX	Zero page	86	ZP	
STX	Zero page,Y	96	ZP	
STY	Absolute	8C	LO	HI
STY	Zero page	84	ZP	
STY	Zero page,X	94	ZP	
TAX	Implied	AA		
TAY	Implied	A8		
TSX	Implied	BA		
TXA	Implied	8A		
TXS	Implied	9A		
TYA	Implied	98		

ROM Kernal Routines

ROM Kernal Routines

Ottis R. Cowper

Standard Commodore Jump Table

ACPTR 65445 $FFA5

This low-level I/O routine retrieves a byte from a serial device without checking for a previous I/O error. If the operation is successful, the accumulator will hold the byte received from the device. The contents of .X and .Y are preserved. The success of the operation will be indicated by the value in the serial status flag upon return. (See READST for details.)

For the routine to function properly, the serial device must currently be a talker on the serial bus, which requires a number of setup steps. Generally, it's preferable to use the higher-level CHRIN routine instead.

CHKIN 65478 $FFC6

This routine specifies a logical file as the source of input in preparation for using the CHRIN or GETIN routines. The logical file should be opened before this routine is called. (See the OPEN routine.) The desired logical file number should be in .X when this routine is called. The contents of .Y are unaffected, but the accumulator value will be changed.

The routine sets the input channel (location $99) to the device number for the specified file. If the device is RS-232 (device number 2), the CIA #2 interrupts for RS-232 reception are enabled. If a serial device (device number 4 or greater) was specified, the device is made a talker on the serial bus.

If the file is successfully set for input, the status-register carry bit will be clear upon return. If carry is set, the operation was unsuccessful and the accumulator will contain a Kernal error-code value indicating which error occurred. Possible error codes include 3 (file was not open), 5 (device did not re-

spond), and 6 (file was not opened for input). The RS-232 and serial status-flag locations also reflect the success of operations for those devices. (See READST for details.)

The JMP to the CHKIN execution routine is by way of the ICHKIN indirect vector at 798–799 ($031E–$031F). You can modify the actions of CHKIN by changing the vector to point to a routine of your own.

CHKOUT 65481 $FFC9

This routine (some Commodore references call it CKOUT) specifies a logical file as the recipient of output in preparation for using the CHROUT routine. The logical file should be opened before this routine is called. (See the OPEN routine.) The desired logical file number should be in .X when this routine is called. The contents of .Y are unaffected, but the accumulator will be changed.

The routine sets the output channel (location $9A) to the device number for the specified file. If the device is RS-232 (device number 2), the routine also enables the CIA #2 interrupts for RS-232 transmission. If a serial device (device number 4 or greater) is specified, the device is also made a listener on the serial bus.

If the file is successfully set for output, the status-register carry bit will be clear upon return. If the carry is set, the operation was unsuccessful, and the accumulator will contain a Kernal error-code value indicating which error occurred. Possible error codes include 3 (file was not open), 5 (device did not respond), and 7 (file was not opened for output). The RS-232 and serial status-flag locations also reflect the success of operations for those devices. (See READST for details.)

The JMP to the CHKOUT execution routine is by way of the ICKOUT indirect vector at $0320–$0321. You can modify the actions of the routine by changing the vector to point to a routine of your own.

CHRIN 65487 $FFCF

This high-level I/O routine (some Commodore references may call it BASIN) receives a byte from the logical file currently specified for input (to change the default input device, see CHKIN above). Except to use the routine to retrieve input from the keyboard when the system is set for default I/O, you must open a logical file to the desired device and specify the file as the input source before calling this routine. (See the OPEN and CHKIN routines.)

For keyboard input (device 0), the routine accepts keypresses until RETURN is pressed, and then returns characters from the input string one at a time on each subsequent call. The character code for RETURN, 13, is returned when the end of an input string is reached. (The Kernal GETIN routine is better for retrieving individual keypresses.)

For tape (device 1), the routine retrieves the next character from the cassette buffer. If all characters have been read from the buffer, the next data block is read from tape into the buffer.

For RS-232 (device 2), the routine returns the next available character from the RS-232 input buffer. If the buffer is empty, the routine waits until a character is received—unless the RS-232 status flag indicates that the DSR signal from the external device is missing, in which case a RETURN character code, 13, is returned.

CHRIN from the screen (device 3) retrieves characters one at a time from the current screen line, ending with a RETURN character code when the last nonspace character on the logical line is reached. (Note that CHRIN from the screen does not work properly in the original version of the 128 Kernal.) For serial devices (device numbers 4 and higher), the routine returns the next available character from the serial bus, unless the serial status flag contains a nonzero value. In that case, the RETURN character code is returned.

For all input devices, the received byte will be in the accumulator upon return. The contents of .X and .Y are preserved during input from the keyboard, screen, or RS-232. For input from tape, only .X is preserved. For input from serial devices, only .Y is preserved. For input from the screen, keyboard, or serial devices, the status-register carry bit will always be clear upon return. For tape input, the carry bit will be clear unless the operation was aborted by pressing the RUN/STOP key. For tape, serial, or RS-232 input, the success of the operation will be indicated by the value in the status-flag location. (See the entry for READST.) The RS-232 portion of the original 128 version of CHRIN has a bug: The carry bit will be set if a byte was successfully received, and will be clear only if the DSR signal is missing—the opposite of the settings for the 64. It's better to judge the success of an RS-232 operation by the value in the status-flag location rather than by the carry-bit setting. (See the READST routine.)

The JMP to the CHRIN execution routine is by way of the

ICHRIN indirect vector at $0324–$0325. You can modify the actions of the routine by changing the vector to point to a routine of your own.

CHROUT 65490 $FFD2

This routine (some Commodore references call it BSOUT) sends a byte to the logical file currently specified for output. Except to send output to the screen when the system is set for default I/O, you must open a logical file to the desired device and specify the file as the output target before calling this routine. (See the OPEN and CHKOUT routines.)

For output to tape (device 1), the character is stored at the next available position in the cassette buffer. When the buffer is full, the data block is written to tape.

For output to RS-232 (device 2), the character is stored in the next available position in the RS-232 output buffer. If the buffer is full, the routine waits until a character is sent.

For output to the screen (device 3), the character is printed at the current cursor position. For serial devices (device numbers 4 and higher), the CIOUT routine is called.

Regardless of the output device, the contents of the accumulator, .X, and .Y are preserved during this routine. The status-register carry bit will always be clear upon return, unless output to tape is aborted by pressing the RUN/STOP key. (In that case, the accumulator will also be set to 0, setting the status-register Z bit as well.) For tape, serial, or RS-232 output, the success of the operation will be indicated by the value in the status flag. (See READST for details.)

The JMP to the CHROUT execution routine is by way of the ICHROUT indirect vector at $0326–$0327. You can modify the actions of the routine by changing the vector to point to a routine of your own.

CINT 65409 $FF81

This routine initializes all RAM locations used by the screen editor, returning screen memory to its default position and setting default screen and border colors. The routine also clears the screen and homes the cursor. All processor registers are affected.

For the 64 only, this routine initializes all VIC chip registers to their default values (that's done during the Kernal IOINIT routine in the 128). For the 128, CINT clears both displays and redirects printing to the display indicated by the position of the 40/80 DISPLAY key. The 128 routine also sets

SID volume to zero and resets programmable function keys to their default definitions. It does not, however, reinitialize the 80-column character set. (That's also part of IOINIT.)

CIOUT 65448 $FFA8

This low-level I/O routine sends a byte to a serial device. The accumulator should hold the byte to be sent. All register values are preserved. The success of the operation will be indicated by the value in the serial status flag. (See READST for details.)

For the routine to function properly, the target serial device must currently be a listener on the serial bus, which requires a number of setup steps. However, if you have already performed all the preparatory steps necessary for CHROUT to a serial device, then you can freely substitute CIOUT for CHROUT, since, for a serial device, CHROUT simply jumps to the CIOUT routine.

CLALL 65511 $FFE7

This routine resets the number of open files (location $98) to zero, then falls through into the CLRCH routine to reset default I/O. The contents of .A and .X are changed, but .Y is unaffected.

Despite its name, the routine doesn't actually close any files that may be open to tape, disk, or RS-232 devices. Unclosed files may cause problems, particularly on disks, so this routine is of limited usefulness. The 128 Kernal provides an alternate routine that does properly close all files open to a serial device. (See CLOSE_ALL.)

The JMP to the CLALL execution routine is by way of the ICLALL indirect vector at $032C–$032D. You can modify the actions of the routine by changing the vector to point to a routine of your own.

CLOSE 65475 $FFC3

This routine closes a specified logical file. Call the routine with the accumulator holding the number of the logical file to be closed. If no file with the specified logical file number is currently open, no action is taken and no error is indicated. If a file with the specified number is open, its entry in the logical file number, device number, and secondary address tables will be removed. For RS-232 files, the driving CIA #2 interrupts will also be disabled. For tape files, the final block of data will be written to tape (followed by an end-of-tape marker, if one was

specified). For disk files, the EOI sequence will be performed.

The 128 version of the routine offers a special close function for disk files: If this routine is called with the status-register carry bit set, and if the device number for the file is 8 or greater, and if the file was opened with a secondary address of 15, then the EOI sequence is skipped. (The table entries for the file are deleted, but that's all.) This solves a problem in earlier versions of the Kernal for disk files opened with a secondary address of 15, the command channel to the drive. An attempt to close the command channel will result in an EOI sequence that closes all files currently open to the drive, not just the command-channel file. This special mode allows the command-channel file to be closed without disturbing other files that may be open to the drive.

The JMP to the CLOSE execution routine is by way of the ICLOSE indirect vector at $031C–$031D. You can modify the actions of the routine by changing the vector to point to a routine of your own.

CLRCHN 65484 $FFCC

This routine restores the default I/O sources for the operating system. The output channel (location $9A) is reset to device 3, the video display. (If the previous output channel was a serial device, it is sent an UNLISTEN command.) The input channel (location $99) is reset to device 0, the keyboard. (If the previous input channel was a serial device, it is sent an UNTALK command.) The contents of .X and .A are changed, but .Y is unaffected.

The JMP to the CLRCHN execution routine is by way of the ICLRCH indirect vector at $0322–$0323. You can modify the actions of the routine by changing the vector to point to a routine of your own.

GETIN 65508 $FFE4

This routine retrieves a single character from the current input device. The routine first checks to see whether the input device number is 0 (keyboard) or 2 (RS-232). If it's not either of these, the Kernal CHRIN routine is called instead. For keyboard or RS-232, the retrieved character will be in the accumulator upon return, and the status-register carry bit will be clear. If no character is available, the accumulator will contain 0. (CHRIN, by contrast, will wait for a character.) The contents of .Y are unaffected, but .X will be changed. For RS-232, bit 3

of the status flag will also be set if no characters are available. (See READST for details.)

The JMP to the GETIN execution routine is by way of the IGETIN indirect vector at \$032A–\$032B. You can modify the actions of the routine by changing the vector to point to a routine of your own.

IOBASE 65523 \$FFF3

This routine returns a constant I/O chip base-address value in .X (low byte) and .Y (high byte). The accumulator is unaffected. For the 64, the value returned is \$DC00—the address of CIA #1. For the 128, the value is \$D000—the address of the VIC chip.

IOINIT 65412 \$FF84

This routine initializes the CIA chips' registers to their default values, along with related RAM locations. All processor registers are affected. For the 128, the routine also initializes the VIC and VDC chip registers (a step which is part of the Kernal CINT routine in the 64). In addition, the 128 routine sets all SID chip registers to zero and calls the Kernal DLCHR routine to initialize the character set for the 80-column chip.

LISTEN 65457 \$FFB1

This low-level serial I/O routine sends a LISTEN command to a specified serial device. Call the routine with the accumulator holding the device number (4–31) of the serial device to receive the command. The contents of .A and .X will be changed; .Y is unaffected. The success of the operation will be indicated by the value in the serial status flag upon return. (See READST for details.)

LOAD 65493 \$FFD5

This routine loads a program file from tape or disk into a specified area of memory, or verifies a program file against the contents of a specified area of memory. A number of preparatory routines must be called before LOAD: SETLFS, SETNAM, and (for the 128 only) SETBNK. See the discussions of those routines for details.

SETLFS establishes the device number and secondary address for the operation. (The logical file number isn't significant for loading or verifying.) The secondary-address value determines whether the load/verify will be absolute or relocating. If bit 0 of the secondary address is %0 (if the value is 0 or any

even number, for example), a relocating load will be performed: The file will be loaded starting at the address specified in .X and .Y. If the bit is %1 (if the value is 1 or any odd number, for example), an absolute load will be performed: The data will be loaded starting at the address specified in the file itself. For tape files, the secondary-address specification can be overridden by the file's internal type specification. Nonrelocatable tape program files always load at their absolute address, regardless of the secondary address.

When calling the LOAD routine, the accumulator should hold the operation type value (0 for a load, or any nonzero value for a verify). If the secondary address specifies a relocating load, the starting address at which data is to be loaded should be stored in .X (low byte) and .Y (high byte). The values of .X and .Y are irrelevant for an absolute load.

The status-register carry bit will be clear upon return if the file was successfully loaded, or set if an error occurred or if the RUN/STOP key was pressed to abort the load. When carry is set upon return, the accumulator will hold a Kernal error-code value indicating the problem. Possible error codes include 4 (file was not found), 5 (device was not present), 8 (no name was specified for a serial load), 9 (an illegal device number was specified).

On the 128 only, the load will be aborted if it extends beyond address $FEFF. This prevents corruption of the MMU configuration register at $FF00. In this case, an error code of 16 will be returned. The success of the operation will also be indicated by the value in the tape/serial status flag. (See READST for details.)

MEMBOT	**65436**	**$FF9C**
MEMTOP	**65433**	**$FF99**

These routines read or set the Kernal's bottom-of-memory pointer and top-of-memory pointer, respectively. (The bottom-of-memory pointer is at locations $0281–$0282 for the 64 or $0A05–$0A06 for the 128; the top-of-memory pointer is at locations $0283–$0284 for the 64 or $0A07–$0A08 for the 128.) To read the pointer, call the routine with the carry flag set; the pointer value will be returned in .X (low byte) and .Y (high byte). To set the pointer, call the routine with the carry flag clear and with .X and .Y containing the low and high bytes, respectively, of the desired pointer value.

OPEN 65472 $FFC0

This routine opens a logical file to a specified device in preparation for input or output. At least one preparatory step is required before the standard OPEN routine is called: SETLFS must be called to establish the logical file number, device number, and secondary address. For tape (device 1), RS-232 (device 2), or serial (device 4 or higher), SETNAM is also required to specify the length and address of the associated filename. For the 128, SETBNK must be called to establish the bank number where the filename can be found.

It is not necessary to load any registers before calling OPEN, and all processor register values may be changed during the routine. The carry will be clear if the file was successfully opened, or it will be set if it could not be opened. When carry is set upon return, the accumulator will hold an error code indicating the problem. Possible error-code values include 1 (ten files—the maximum allowed—are already open), 2 (a currently open file already uses the specified logical file number), and 5 (specified device did not respond). The RS-232 and tape/serial status flags will also reflect the success of the operation for those devices. (See READST for details.)

On the 128, there is an exception to the carry-bit rule. Because of a bug in the 128's RS-232 OPEN routine, carry will be set if the RS-232 device is present when x-line handshaking is used (if the DSR line is high), or clear if the device is absent—the opposite of the proper setting.

The JMP to the OPEN execution routine is by way of the IOPEN indirect vector $031A–$031B. You can modify the actions of the routine by changing the vector to point to a routine of your own.

PLOT 65520 $FFF0

This routine reads or sets the cursor position on the active display. If it is called with the status-register carry bit clear, the value in .X specifies the new cursor row (vertical position), and the value in .Y specifies the column (horizontal position). The carry bit will be set upon return if the specified column or row values are beyond the right or bottom margins of the current output window, or it will be clear if the cursor was successfully positioned.

If the routine is called with the carry bit set, the row number for the current cursor position is returned in .X and the current column number is returned in .Y. For the Commodore

128, the cursor position will be relative to the home position of the current output window rather than to the upper left corner of the screen. Of course, in the case of a full-screen output window—the default condition—the upper left corner of the screen is the home position of the window.

RAMTAS 65415 $FF87

This routine clears zero-page RAM (locations $02–$FF) and initializes Kernal memory pointers in zero page. For the 64 only, the routine also clears pages 2 and 3 (locations $0200–$03FF), tests all RAM locations from $0400 upwards until ROM is encountered, and sets the top-of-memory pointer. For the 128, the routine sets the BASIC restart vector ($0A00) to point to BASIC's cold-start entry address, $4000.

RDTIM 65502 $FFDE

This routine returns the current value of the jiffy clock. The clock value corresponds to the number of jiffies (1/60-second intervals) that have elapsed since the system was turned on or reset, or the number of jiffies since midnight if the clock value has been set. The low byte of the clock value (location $A2) is returned in .A, the middle byte (location $A1) in .X, and the high byte (location $A0) in .Y.

READST 65463 $FFB7

This routine (some Commodore references call it READSS) returns the status of the most recent I/O operation. The status value will be in the accumulator upon return; the contents of .X and .Y are unaffected. If the current device number is 2 (indicating an RS-232 operation), the status value is retrieved from the RS-232 status flag (location $0297 for the 64 or $0A14 for the 128), and the flag is cleared. Otherwise, the status value is retrieved from the tape/serial status flag (location $90). That flag is not cleared after being read.

Bit	Value	Meaning if set Serial	Meaning if set Tape	Meaning if set RS-232
0	1/$01	write timeout		parity error
1	2/$02	read timeout		framing error
2	4/$04		short block	receiver buffer overflow
3	8/$08		long block	receiver buffer empty
4	16/$10	verify mismatch	unrecoverable read or verify mismatch	CTS missing
5	32/$20		checksum mismatch	
6	64/$40	EOI (end of file)	end of file	DSR missing
7	128/$80	device not present	end of tape	break

RESTOR **65418** **$FF8A**

This routine resets the Kernal indirect vectors ($0314–$0333) to their default values. All processor registers are affected.

SAVE **65496** **$FFD8**

This routine saves the contents of a block of memory to disk or tape. It could be a BASIC or ML program, but it doesn't have to be. A number of preparatory routines must be called first: SETLFS, SETNAM, and (for the 128 only) SETBNK. See the discussions of those routines for details.

SETLFS establishes the device number and secondary address for the operation. (The logical file number isn't significant for saving.) The secondary address is irrelevant for saves to serial devices, but for tape it specifies the header type. If bit 0 of the secondary address value is %1 (if the value is 1, for example), the data will be stored in a nonrelocatable file—one that will always load to the same memory address from which it was saved. Otherwise, the data will be stored in a file that can be loaded to another location. If bit 1 of the secondary address is %1 (if the value is 2 or 3, for example), the file will be followed by an end-of-tape marker.

Before calling SAVE, you must also set up a two-byte zero-page pointer containing the starting address of the block of memory to be saved and then store the address of the zero-page pointer in the accumulator. The ending address (plus one) for the save should be stored in .X (low byte) and .Y (high byte). To save the entire contents of the desired area, it's important to remember that .X and .Y must hold an address that is one location beyond the desired ending address.

When the save is complete, the carry will be clear if the file was successfully saved, or set if an error occurred (or if the RUN/STOP key was pressed to abort the save). When carry is set upon return, the accumulator will hold the Kernal error code indicating the problem. Possible error-code values include 5 (serial device was not present), 8 (no name was specified for a serial save), and 9 (an illegal device number was specified). The success of the operation will also be indicated by the value in the tape/serial status flag. (See READST for details.)

SCNKEY **65439** **$FF9F**

This routine scans the keyboard matrix to determine which keys, if any, are currently pressed. The standard IRQ service

routine calls SCNKEY, so it's not usually necessary to call it explicitly to read the keyboard. The character code for the key currently pressed is loaded into the keyboard buffer, from where it can be retrieved using the Kernal GETIN routine. The matrix code of the keypress read during this routine can also be read in location $CB (64) or $D4 (128), and the status of the shift keys can be read in location $028D (64) or $D3 (128).

SCREEN 65517 $FFED

This routine (Commodore 128 literature calls it SCRORG) returns information on the size of the screen display. For the 64, the routine always returns the same values—the screen width in columns (40) in .X and the screen height in rows (25) in .Y. The accumulator is unaffected. For the 128, the values returned reflect the size of the current output window. The X register will contain in the current window the number of columns minus one, and .Y will contain the number of rows minus one. The accumulator will hold the maximum column number for the display currently active (39 for the 40-column screen or 79 for the 80-column screen).

SECOND 65427 $FF93

This low-level serial I/O routine sends a secondary address to a device which has been commanded to listen. The value in the serial status flag upon return will indicate whether the operation was successful.

SETLFS 65466 $FFBA

This routine assigns the logical file number (location $B8), device number (location $BA), and secondary address (location $B9) for the current I/O operation. Call the routine with the accumulator holding the logical file number, .X holding the device number, and .Y holding the secondary address. All register values are preserved during the routine. Refer to the LOAD and SAVE routines for the special significance of the secondary address in those cases. When OPENing files to serial devices, it's vital that each logical file have a unique secondary address. In the 128 Kernal, the LKUPLA and LKUPSA routines can be used to find unused logical file numbers and secondary addresses.

SETMSG 65424 $FF90

SETMSG sets the value of the Kernal message flag (location $9D). Call the routine with the accumulator holding the de-

sired flag value (.X and .Y are unaffected.) Valid flag values are 0 (no Kernal messages are displayed), 64 (only error messages are displayed), 128 (only control messages—PRESS PLAY ON TAPE, for example—are displayed), and 192 (both error and control messages are displayed).

SETNAM **65469** **$FFBD**

This routine assigns the length (location $B7) and address (locations $BB–$BC) of the filename for the current I/O operation. Call the routine with the length of the filename in .A and the address of the first character of the name in .X (low byte) and .Y (high byte). If no name is used for the current operation, load the accumulator with 0; the values in .X and .Y are then irrelevant. All register values are preserved during this routine.

SETTIM **65499** **$FFDB**

This routine sets the value in the software jiffy clock. The value in the accumulator is transferred to the low byte (location $A2), the value in .X to the middle byte (location $A1), and the value in .Y to the high byte (location $A0). The specified value should be less than $4F1A01, which corresponds to 24:00:00 hours.

SETTMO **65442** **$FFA2**

The SETTMO routine stores the contents of the accumulator in the IEEE timeout flag. (.X and .Y are unaffected.) This routine is superfluous, since the flag isn't used by any 64 or 128 ROM routine. It is present merely to maintain consistency with previous versions of the Kernal. For the 64, the flag location is $0285; for the 128, it's at $0A0E.

STOP **65505** **$FFE1**

This routine checks whether the RUN/STOP key is currently pressed. It returns with the status-register Z bit clear if the key is not pressed, or with the bit set if it is pressed. Additionally, if RUN/STOP is pressed the CLRCH routine is called to restore default I/O channels, and the count of keys in the keyboard buffer is reset to zero.

The JMP to the STOP execution routine is by way of the ISTOP indirect vector at $0328–$0329. You can modify the actions of the routine by changing the vector to point to a routine of your own.

TALK **65460** **$FFB4**

This low-level I/O routine sends a TALK command to a serial device. Call the routine with the accumulator holding the number (4–31) of the device. The success of the operation will be indicated by the value in the serial status flag upon return. (See READST for details.)

TKSA **65430** **$FF96**

This low-level serial I/O routine sends a secondary address to a device which has previously been commanded to talk. The success of the operation will be indicated by the value in the serial status flag upon return. (See READST for details.)

UDTIM **65514** **$FFEA**

This routine increments the software jiffy clock and scans the keyboard column containing the RUN/STOP key. (The 128 version of the routine also decrements a countdown timer.) This routine is normally called every 1/60 second as part of the standard IRQ service routine.

UNLSN **65454** **$FFAE**

This low-level I/O routine sends an UNLISTEN command to all devices on the serial bus. Any devices which are currently listeners will cease accepting data.

UNTLK **65451** **$FFAB**

This low-level I/O routine sends an UNTALK command to all devices on the serial bus. Any devices which are currently talkers will cease sending data.

VECTOR **65421** **$FF8D**

This routine can be used either to store the current values of Kernal indirect vectors at $0314–$0333 or to write new values to the vectors. When calling this routine, .X and .Y should be loaded with the address of a 32-byte table (low byte in .X, high byte in .Y). If the status-register carry bit is clear when the routine is called, the vectors will be loaded with the values from the table. If carry is set, the 16 two-byte address values currently in the vectors will be copied to the table.

New 128 Kernal Jump Table

Locations $FF47-$FF7F comprise a new table of jump vectors to routines found in Commodore 128 ROM, but not in the Commodore 64.

BOOT_CALL **65363** **$FF53**

This routine attempts to load and execute boot sectors from a specified disk drive. Call the routine with .X holding the device number for the drive (usually 8) and with the accumulator holding the character code corresponding to the drive number—not the actual drive number. The single drive in 1541 and 1571 units is drive 0; in this case, use 48, the character code for zero. If the specified drive is not present or is turned off, or if the disk in the drive does not contain a valid boot sector, the routine will return with the status-register carry bit set. If a boot sector is found, it will be loaded into locations $0B00–$0BFF. Additional boot sectors may be loaded into other areas of memory, and the boot code may not return to this routine.

CLOSE_ALL **65354** **$FF4A**

This routine closes all files currently opened to a specified device, providing an improved version of CLALL. Enter the routine with the accumulator holding the number of the device on which files are to be closed. If the specified device is the current input or output device, the input or output channel will be reset to the default device (screen or keyboard). If all files to the device were successfully closed, the status-register carry bit will clear upon return. A set carry bit indicates that a device error occurred.

C64_MODE **65357** **$FF4D**

This is the equivalent of the BASIC command GO 64. It performs an immediate cold start of 64 mode. To get back to 128 mode, it is necessary to reset the computer, or to turn it off and back on.

DLCHR **65378** **$FF62**

This routine copies character shape data for both standard ROM character sets into the VDC video chip's private block of RAM, providing character definitions for the 80-column display. (The VDC has no character ROM.) This routine is also called as part of IOINIT for the 128.

DMA_CALL **65360** **$FF50**

This routine passes a command to a DMA (Direct Memory Access) device. The DMA device will then take control of the system to execute the command. The routine is written to support the REC (RAM Expansion Controller) chip in the 1700

and 1750 Memory Expansion Modules, the only DMA peripherals currently available. Call the routine with .Y holding the command for the DMA device and .X holding the bank number for the operation. Other preparatory steps may be required, depending on the command.

GETCFG **65387** **$FF6B**

This routine translates a bank number (0–15) into the corresponding MMU register setting to configure the system for that bank. Call the routine with .X holding the bank number. Upon return, the accumulator will hold the corresponding MMU configuration register value. (.Y is unaffected.) Once you have this value, you can store it into $FF00 to change banks. The input bank number is not checked for validity, and a number outside the acceptable range will return a meaningless value.

INDCMP **65402** **$FF7A**

This routine compares .A to the number held in a memory location in a specified bank. In preparing to call INDCMP, load a two-byte zero-page pointer with the address of the location with which the accumulator is to be compared (or with the base location if a series of bytes is to be compared), then store the address of this pointer in location $02C8. Call the routine with the accumulator holding the byte to be compared, .X holding the bank number (0–15) for the target location, and .Y holding an offset value which will be added to the address in the pointer. (Load .Y with 0 if no offset is desired.) Upon return, the accumulator will still hold the byte value, and the status-register N, Z, and C (carry) bits will reflect the result of the comparison. The value in .Y will also be preserved, but it is necessary to reload .X with the bank number before every call to this routine. You can compare up to 256 sequential locations without changing the address in the zero-page pointer by simply incrementing .Y between calls.

INDFET **65396** **$FF74**

This routine reads the contents of a location in a specified bank. Prior to calling this routine, you must load a two-byte zero-page pointer with the address of the location to be read (or with the base location if a series of bytes is to be read).

Call the routine with the accumulator holding the address of the zero-page pointer, .X holding the bank number (0–15) for the target location, and .Y holding an offset value which

will be added to the address in the pointer. (Load .Y with 0 if no offset is desired.) Upon return, the accumulator will hold the byte from the specified address. The value in .Y is not changed.

To read from a series of locations, it is necessary to reload the accumulator and .X values before every call to this routine, but you can read up to 256 sequential locations without changing the address in the zero-page pointer by incrementing .Y between calls.

INDSTA 65399 $FF77

This routine stores a value at an address in a specified bank. Before calling the routine, you must load a two-byte zero-page pointer with the address of the location at which the byte is to be stored (or with the base location if a series of bytes is to be stored), and then store the address of this pointer in location $02B9. Call the routine with the accumulator holding the byte to be stored, .X holding the bank number (0–15) for the target location, and .Y holding an offset value which will be added to the address in the pointer. (Load Y with 0 if no offset is desired.) Upon return, the accumulator will still hold the byte value; .Y is also preserved. To write to a series of locations, you must reload .X with the bank number before every call, but you can write to up to 256 sequential locations without changing the address in the zero-page pointer by simply incrementing .Y between calls.

JMPFAR 65393 $FF71

JMPFAR jumps to a routine in a specified bank, with no return to the calling bank. Prior to calling this routine, you must store the bank number (0–15) of the target routine in location 2 and the address of the target routine in locations 3–4 in high-byte/low-byte order, opposite from the usual arrangement. Load location 5 with the value you want placed in the status register when the target routine is entered. (The behavior of many operating-system routines is influenced by the status-register setting, particularly the state of the carry bit. Load 5 with the value 0 to clear carry or with 1 to set carry.) To pass other register values, store the desired accumulator value in location 6, the value for .X in 7, and the value for .Y in 8.

JSRFAR 65390 $FF6E

This routine jumps to a subroutine in a specified bank and returns to the calling routine in bank 15. Prior to calling this

routine, you must store the bank number (0–15) of the target routine in location 2 and the address of the target routine in locations 3–4 (in high-byte/low-byte order, opposite from the usual arrangement). Load location 5 with the value you want placed in the status register when the target routine is called. (The behavior of many operating system routines is influenced by the status-register setting, particularly the state of the carry bit. Load 5 with the value 0 to clear carry, or with 1 to set carry.) To pass other register values to the routine you will be calling, store the desired accumulator value in location 6, the value for .X in 7, and the value for .Y in 8. Upon return, location 5 will hold the status-register value at the time of exit, 6 will hold the accumulator value, 7 will hold the .X value, 8 will hold the .Y value, and 9 will hold the stack-pointer value. The system is always configured for bank 15 upon exit.

LKUPLA 65369 $FF59

This routine checks whether a specified logical file number is currently used. Call the routine with the accumulator holding the logical-file-number value in question. If that file number is available, the carry bit will be set upon return. (The logical file number will still be in the accumulator.) However, if the number is used for a currently open file, then the carry bit will be clear upon return, the accumulator will still hold the logical file number, .X will hold the corresponding device number, and .Y will hold the corresponding secondary address.

LKUPSA 65372 $FF5C

This routine checks whether a specified secondary address is currently in use. Call the routine with .Y holding the secondary-address value in question. If that secondary address is not currently used, the status-register carry bit will be set upon return. (The secondary-address value will still be in .Y.) However, if the number is used for a currently open file, the carry bit will be clear upon return, .Y will still hold the secondary address, the accumulator will hold the associated logical file number, and .X will hold the corresponding device number.

PFKEY 65381 $FF65

When you turn on the 128, its function keys are predefined. Pressing F3 prints DIRECTORY, F7 holds the LIST command, and so on. The PFKEY Kernal routine assigns a new definition to one of the 10 programmable function keys (F1–F8, SHIFT–RUN/STOP, and HELP).

Call the routine with the accumulator holding the address of a three-byte zero-page string descriptor, .X holding the key number (1–10), and .Y holding the length of the new definition string. The first two bytes of the descriptor in zero page should contain the address of the definition string (in the usual low-byte/high-byte order); the final byte should hold the bank number where the definition string is located. PFKEY doesn't check the key number for validity; a value outside the acceptable range may garble existing definitions. Upon return, the carry bit will be clear if the new definition was successfully added, or set if there was insufficient room in the definition table for the new definition.

PHOENIX 65366 $FF56

This routine initializes function ROMs and attempts to boot a disk from the default drive. The presence of function ROMs in cartridges or in the 128's spare ROM socket is recorded during the power-on/reset sequence. This routine initializes the function ROMs by calling their recorded cold-start entry addresses. If ROMs are present, they may or may not return to this routine, depending on the initialization steps performed. If no ROMs are present, or if all ROMs return after initialization, the routine attempts to boot a disk in drive 0 of device 8 using the BOOT_CALL routine.

PRIMM 65405 $FF7D

This routine prints the string of character codes which immediately follows the JSR to this routine. (You must always call this routine with JSR, never with JMP. Only JSR places the required address information on the stack.) The routine continues printing bytes as character codes until a byte containing zero is encountered. When the ending marker is found, the routine returns to the address immediately following the zero byte. All registers (.A, .X, and .Y) are preserved during this routine.

SETBNK 65384 $FF68

This Kernal routine establishes the current memory bank from which data will be read or to which data will be written during load/save operations, as well as the bank where the filename for the I/O operations can be found. Call the routine with the accumulator holding the bank number for data and .X holding the bank for the filename. All registers (.A, .X, and .Y) are preserved during this routine.

SPIN_SPOUT **65351** **$FF47**

This low-level serial I/O routine sets up the serial bus for fast (burst mode) communications. Unless you're writing a custom data-transfer routine, it's not necessary to call this routine explicitly. All higher-level serial I/O routines already include this setup step. The routine should be called with the status-register carry bit clear to establish fast serial input or with the bit set to establish fast serial output.

SWAPPER **65375** **$FF5F**

This routine switches active screen displays. The active display is the one which has a live cursor, and to which screen CHROUT output is directed. The routine exchanges the active and inactive screen-editor variable tables, tab-stop bitmaps, and line-link bitmaps; and it toggles the active screen flag (location $D7). The routine doesn't physically turn either video chip on or off—both chips always remain enabled.

The Routines

Name

Add two bytes and store the result

Description

Adding is one of the essential arithmetic functions in machine language (or in any computer language). This routine simply adds two numbers and stores the result in memory.

Prototype

1. Load the first number from memory.
2. Clear the carry flag with a CLC instruction.
3. Add the second number with ADC.
4. Save the result in memory.

Explanation

The framing routine waits for a keypress, then stores the ASCII value in memory. It gets a second ASCII value, then prints the two numbers. After the **ADDBYT** routine is called, the answer is printed.

If you want a proper result, you should always clear carry before using the ADC instruction. ADC really adds three numbers: two that are in the range 0–255 and one (the carry flag) that's either 0 or 1. Adding 10 + 10 with carry set (10 + 10 + 1) will give you a result of 21.

Note: If the result of the addition is greater than 255, the additional bit which represents a value of 256 will be in the carry flag (carry will be set). If you're adding signed bytes and the answer is greater than 127, the overflow (V) flag will be set.

Routine

```
C000                GETIN   =    $FFE4
C000                LINPRT  =    $BDCD       ; LINPRT = $8E32 on the 128
C000                CHROUT  =    $FFD2
                                             ;
C000  20 37 C0              JSR  GETKEY      ; get a key (ASCII value)
C003  8D 3D C0              STA  NUMBER1     ; store it
C006  20 37 C0              JSR  GETKEY      ; get a second key
C009  8D 3E C0              STA  NUMBER2     ; store it, too
C00C  AE 3D C0              LDX  NUMBER1     ; now print it
C00F  A9 00                 LDA  #0
C011  20 CD BD              JSR  LINPRT
C014  A9 0D                 LDA  #13
C016  20 D2 FF              JSR  CHROUT      ; print <RETURN>
C019  AE 3E C0              LDX  NUMBER2     ; second number
C01C  A9 00                 LDA  #0
C01E  20 CD BD              JSR  LINPRT      ; print it
C021  A9 0D                 LDA  #13
C023  20 D2 FF              JSR  CHROUT      ; <RETURN> again
                                             ;
C026  AD 3D C0      ADDBYT  LDA  NUMBER1     ; the first number
C029  18                    CLC              ; clear the carry flag
```

```
C02A  6D  3E  C0                 ADC   NUMBER2     ; add the second
C02D  8D  3F  C0                 STA   TOTAL       ; store it
                                                   ;
C030  AA                         TAX               ; put it in .X
C031  A9  00                     LDA   #0
C033  20  CD  BD                 JSR   LINPRT      ; and print it
C036  60                         RTS
                                                   ;
C037  20  E4  FF   GETKEY        JSR   GETIN
C03A  F0  FB                     BEQ   GETKEY
C03C  60                         RTS
                                                   ;
C03D  00           NUMBER1       .BYTE 0
C03E  00           NUMBER2       .BYTE 0
C03F  00           TOTAL         .BYTE 0
```

See also ADDFP, ADDINT, INC2.

Name

Add two floating-point numbers using the ROM routine

Description

Enter this routine with the two numbers to be added in the floating-point accumulators FAC1 and FAC2. The ROM routine FADDT then adds them together and returns the answer in FAC1.

Prototype

1. Store one number in FAC1.
2. Store the other in FAC2.
3. Call FADDT.

Explanation

Like most of the other floating-point routines in this book, **ADDFP** depends on built-in ROM routines. The framing program starts by converting the integer 15 to floating-point format, via GIVAYF. Next, MOVEF moves the number from FAC1 to FAC2. GIVAYF converts another integer—1325—to floating-point.

The numbers are added in **ADDFP** which simply calls FADDT. Back in the main routine, FOUT converts FAC1 to a printable ASCII format, and the result is printed to the screen.

Routine

```
C000                ZP       =    $FB
C000                CHROUT   =    $FFD2
C000                FADDT    =    $B86A     ; FADDT = $8848 on the 128—adds FAC1
                                            ; to FAC2; result in FAC1
C000                MOVEF    =    $BC0F     ; MOVEF = $8C3B on the 128—moves
                                            ; FAC1 to FAC2
C000                GIVAYF   =    $B391     ; GIVAYF = $AF03 on the 128—converts
                                            ; integer to floating point
C000                FOUT     =    $BDDD     ; FOUT = $8E42 on the 128—converts FAC1
                                            ; to ASCII string
                                            ;
                                            ; Convert the numbers 15 and 1325 to
                                            ; floating point and add them.
C000  A9 00                   LDA  #>15     ; high byte of 15
C002  A0 0F                   LDY  #<15     ; low byte
C004  20 91 B3                JSR  GIVAYF   ; convert it. now it's in FAC1
C007  20 0F BC                JSR  MOVEF    ; move FAC1 to FAC2
C00A  A9 05                   LDA  #>1325   ; high byte of 1325
C00C  A0 2D                   LDY  #<1325   ; low byte
C00F  20 91 B3                JSR  GIVAYF   ; convert it
                                            ; FAC1 now holds 1325, and FAC2 holds 15.
C011  20 29 C0                JSR  ADDFP    ; add them
C014  20 DD BD                JSR  FOUT     ; convert to ASCII
C017  85 FB                   STA  ZP       ; pointer
```

```
C019  84  FC             STY   ZP+1        ; to the string
C01B  A0  00             LDY   #0
C01D  B1  FB     PRTLOP  LDA   (ZP),Y
C01F  D0  01             BNE   PRNIT
C021  60                 RTS
C022  20  D2 FF  PRNIT   JSR   CHROUT
C025  C8                 INY
C026  D0  F5             BNE   PRTLOP
C028  60                 RTS
                                           ;
C029  20  6A B8  ADDFP   JSR   FADDT       ; add FAC1 and FAC2
C02C  60                 RTS               ; the result is in FAC1
```

See also ADDBYT, ADDINT, INC2.

Name

Add two 2-byte integer values and store the result in memory

Description

Adding two integers is a matter of clearing the carry flag and then using the ADC (ADd with Carry) instruction, first on the low byte and then on the high byte.

Prototype

1. Clear the carry flag.
2. Load the low byte of the first number into .A.
3. Add the low byte of the second number and store the result.
4. Repeat by adding the high bytes of the two numbers.

Explanation

Adding multiple-byte numbers is reasonably easy. The important thing is to start with the low byte and work your way up to the higher bytes. Remember the convention that low bytes are stored in memory before the high bytes. The number 1000 is hex $03E8, which would be stored as an $E8 followed by an $03.

For each byte, addition is a three-step process: Load the first number (LDA), add the second (ADC), and store the result somewhere (STA). Also, carry should be cleared before the first byte is added. After that, carry handles itself.

The following program starts with the number 1000 and loops 30 times, repeatedly adding 350 to the total in NUM1. After each step, the current value is printed to the screen.

Routine

```
C000                LINPRT  =    $BDCD      ; LINPRT = $8E32 on the 128
C000                CHROUT  =    $FFD2
                                            ;
                                            ; Start at 1000 and add 350, repeating 30
                                            ; times.
C000  A9 E8                 LDA  #<1000     ; set up NUM1
C002  8D 47 C0              STA  NUM1       ; with the low byte
C005  A9 03                 LDA  #>1000     ; and high byte
C007  8D 48 C0              STA  NUM1+1
C00A  A9 5E                 LDA  #<350      ; NUM2 needs
C00C  8D 49 C0              STA  NUM2       ; a low byte
C00F  A9 01                 LDA  #>350      ; and
C011  8D 4A C0              STA  NUM2+1     ; a high byte
                                            ;
C014  A9 1E                 LDA  #30        ; the counter
C016  8D 4B C0              STA  RPT        ; is stored in RPT (number of repetitions)
C019  20 2A C0  LOOP        JSR  PRNNUM     ; print the number
C01C  A9 20                 LDA  #32        ; space character
C01E  20 D2 FF              JSR  CHROUT     ; print it
```

```
C021  20  33  C0            JSR   ADDINT      ; add NUM2 to NUM1
C024  CE  4B  C0            DEC   RPT         ; RPT counts down
C027  D0  F0                BNE   LOOP        ; and loop back for more
C029  60                    RTS               ; finished
                                              ;
C02A  AE  47  C0  PRNNUM    LDX   NUM1        ; low byte of NUM1
C02D  AD  48  C0            LDA   NUM1+1      ; high byte
C030  4C  CD  BD            JMP   LINPRT      ; print it (RTS is implied)
                                              ;
C033  18          ADDINT    CLC               ; always clear carry before adding
C034  AD  49  C0            LDA   NUM2        ; low byte of NUM2
C037  6D  47  C0            ADC   NUM1        ; add to low byte of NUM1
C03A  8D  47  C0            STA   NUM1        ; store it
                                              ; Now carry is indeterminate, but it's
                                              ; handled by the ADC below.
                                              ; Note that you don't CLC before adding
                                              ; the high byte.
C03D  AD  4A  C0            LDA   NUM2+1      ; high byte
C040  6D  48  C0            ADC   NUM1+1      ; add it
C043  8D  48  C0            STA   NUM1+1      ; store it
C046  60                    RTS               ; done
                                              ;
C047  00  00      NUM1      .BYTE 0,0
C049  00  00      NUM2      .BYTE 0,0
C04B  00          [illegible]  .BYTE 0
```

See also ADDBYT, ADDFP, INC2.

Name

Set up a time-of-day (TOD) alarm

Description

Both CIA time-of-day clocks are equipped with a built-in alarm function. To use the alarm, you must set both the clock and the alarm time, just as you would on any alarm clock. Rather than actually sounding a tone when the clock time matches the alarm time, the TOD clock triggers an interrupt. Your program must then take appropriate action, depending upon the intended use of the alarm.

A TOD alarm can be used in any number of ways. In an arcade-style game, it can signal the end of one player's turn, the completion of a particular skill level, or the end of the game itself. In an educational program, the alarm can signal when the user has taken too much time to respond.

The alarm mechanisms on the two TOD clocks are practically identical. The only difference is that, because of the way the CIA chips are wired into the system, TOD clock 1 causes an IRQ interrupt while TOD clock 2 triggers an NMI interrupt. In **ALARM2**, we produce a tone when the second TOD clock alarm causes such an interrupt.

Prototype

In **ALARM2**:

1. Store the current time in binary-coded decimal (BCD) format as TIMSET at the end of the program.
2. Define the alarm time in BCD format as ALARTM1.
3. Redirect the NMI interrupt vector at 792 to MAIN.
4. Set bit 7 of control register B at 56591 (CI2CRB) and set the alarm time for TOD clock 2 using ARMTIM.
5. Then clear this bit and set the current time for TOD clock 2, again using ARMTIM.
6. Set bit 2, the alarm interrupt bit, in the interrupt control register (CI2ICR) at 56589 and RTS. Bit 7 must be set in order to set bit 2.

In MAIN:

1. Determine whether the alarm caused the NMI interrupt by testing bit 7 of the interrupt control register (CI2ICR).
2. If this bit is clear, exit the routine through the normal NMI interrupt handler (in step 7).
3. Otherwise, clear the alarm bit (bit 2) in CI2ICR. Bit 7 must be set to zero in order to clear this bit.

4. Set the parameters of the SID chip to produce an alarm sound and start the attack/decay/sustain cycle of the chip.
5. Wait for a keypress with SCNKEY, a Kernal routine.
6. When a keypress occurs, stop the alarm sound by clearing the SID chip, restore the normal NMI vector address, and clear the keyboard buffer.
7. Exit the routine by executing the normal NMI interrupt handler.

Explanation

When **ALARM2** ($C000–$C009) is set up, the NMI interrupt vector is changed so that it points to our own routine at MAIN. Next, with the subroutine ARMTIM, we set the TOD clock time to 4:05:10.0 p.m. and the alarm time to three seconds later, or 4:05:13.0 p.m.

ARMTIM is similar to **TOD2ST**, which sets the second TOD clock. In **TOD2ST**, .Y is always initialized to 0, whereas in ARMTIM, .Y is initially 0 or 4. This allows you to set either the TOD time or the alarm time with the same routine. If .Y is 0, the alarm time, defined as ALARTM, is set. If .Y is 4, the TOD clock time, or TIMSET, is set.

ALARTM and TIMSET can be set to any times you like. Both are expressed in binary-coded decimal (BCD) format.

Before the setup routine is exited, the TOD alarm interrupt is enabled by setting bit 2 of the interrupt control register (CI2ICR). Notice that bit 7 of this register must be set in order to set bits 0–6. To clear one of these bits, store a zero in bit 7 while storing a one in the bit you wish to clear.

Having now pointed the NMI vector to our own routine, the first thing the computer does when an NMI interrupt occurs in MAIN is to check to see whether our alarm caused this interrupt. If the NMI interrupt has been caused by another source, the normal NMI interrupt handler is accessed. Otherwise, the alarm interrupt is disabled, and the current alarm action is carried out—in this case, sounding a tone until a key is pressed.

Once the SID chip starts the tone, we rely on the Kernal routine SCNKEY rather than GETIN to check for a keypress. SCNKEY, unlike GETIN, works during interrupts.

When you finally press a key, the SID chip is turned off with **SIDCLR**, and the normal NMI vector is restored with **RSTVEC**.

Note: **ALARM2** demonstrates how to use TOD clock 2, on CIA (Complex Interface Adapter) chip 2, to signal an alarm. But if you're already using the second TOD clock elsewhere in your program, the first TOD clock will work equally well in this capacity.

To set up the alarm on TOD clock 1, use the equivalent TOD registers (TODTN1) and interrupt control registers (CIAICR, CIACRB) found in CIA 1 (each of these is lower in memory by 256 bytes). Since the interrupt generated by TOD clock 1 is an IRQ interrupt, redirect the IRQ interrupt vector at 788, rather than the NMI vector, to your custom routine

Routine

```
C000               TODTN2   =    56584         ; time-of-day clock 2—tenths-of-seconds
                                               ; register
C000               RESTOR   =    65418         ; routine to restore Kernal vectors
C000               NMIVEC   =    792           ; vector to NMI interrupt routine
C000               NMINOR   =    65095         ; NMINOR = 64064 on the 128—normal
                                               ; NMI interrupt service routine
C000               CI2CRB   =    56591         ; CIA 2 control register B
C000               CI2ICR   =    56589         ; CIA 2 interrupt control register
C000               SIGVOL   =    54296         ; SID chip volume register
C000               ATDCY1   =    54277         ; voice 1 attack/decay register
C000               SUREL1   =    54278         ; voice 1 sustain/release register
C000               FREHI1   =    54273         ; voice 1 frequency control (high byte)
C000               FRELO1   =    54272         ; voice 1 frequency control (low byte)
C000               VCREG1   =    54276         ; voice 1 control register
C000               SCNKEY   =    65439         ; Kernal routine to get a keypress
C000               NDX      =    198           ; NDX = 208 on the 128—number of
                                               ; characters in keyboard buffer
                                               ;
                                               ; Set up an alarm clock signal using TOD
                                               ; clock 2.
C000  A9 2A        ALARM2   LDA  #<MAIN        ; store the low byte of NMI interrupt
                                               ; wedge
C002  8D 18 03              STA  NMIVEC
C005  A9 C0                 LDA  #>MAIN        ; and the high byte
C007  8D 19 03              STA  NMIVEC+1
C00A  AD 0F DD              LDA  CI2CRB        ; get current register value
C00D  09 80                 ORA  #%10000000    ; turn on bit 7 to set alarm time
C00F  8D 0F DD              STA  CI2CRB
C012  A0 00                 LDY  #0            ; to index alarm time setting
C014  20 66 C0              JSR  ARMTIM        ; set TOD clock 2 alarm time
C017  AD 0F DD              LDA  CI2CRB        ; now, clear bit 7 of the control register to
                                               ; set TOD time
C01A  29 7F                 AND  #%01111111    ; turn off bit 7
C01C  8D 0F DD              STA  CI2CRB
C01F  A0 04                 LDY  #4            ; to index the time setting
C021  20 66 C0              JSR  ARMTIM        ; set the TOD 2 time
C024  A9 84                 LDA  #%10000100    ; set bits 2 and 7 to enable TOD alarm
                                               ; interrupt
C026  8D 0D DD              STA  CI2ICR
C029  60                    RTS                ; exit setup routine
                                               ;
C02A  AD 0D DD     MAIN     LDA  CI2ICR        ; did the alarm cause the interrupt (is bit 2
                                               ; set?)?
C02D  29 04                 AND  #%00000100
C02F  F0 32                 BEQ  EXIT          ; bit 2 is clear, so execute normal interrupts
```

```
C031  A9 04              LDA   #%00000100    ; the alarm triggered the interrupt, so clear
                                             ; the alarm bit
C033  8D 0D DD           STA   CI2ICR
                                             ; And signal with an alarm sound.
C036  20 73 C0           JSR   SIDCLR        ; clear the SID chip
C039  A9 0D              LDA   #13           ; set the volume
C03B  8D 18 D4           STA   SIGVOL
C03E  A9 00              LDA   #$0           ; set attack/decay
C040  8D 05 D4           STA   ATDCY1
C043  A9 F0              LDA   #$F0          ; set sustain/release
C045  8D 06 D4           STA   SUREL1
C048  A9 04              LDA   #4            ; set voice 1 high frequency
C04A  8D 01 D4           STA   FREHI1
C04D  A9 21              LDA   #%00100001    ; select sawtooth waveform and gate the
                                             ; sound
C04F  8D 04 D4           STA   VCREG1
C052  20 9F FF  WAIT     JSR   SCNKEY        ; wait for a keypress
C055  A5 C6              LDA   NDX           ; check keyboard buffer
C057  F0 F9              BEQ   WAIT          ; if no key is pressed, wait
C059  20 73 C0           JSR   [redacted]    ; stop the alarm sound
C05C  20 7E C0           JSR   RSTVEC        ; restore NMI vector
C05F  A9 00     BUFCLR   LDA   #0            ; clear keyboard buffer
C061  85 C6              STA   NDX
C063  4C 47 FE  EXIT     JMP   NMINOR        ; exit through normal NMI interrupt
                                             ; handler
                                             ;
                                             ; Set alarm and time. Come in with .Y = 0
                                             ; to set alarm and .Y = 4 to set time.
C066  A2 03     ARMTIM   LDX   #3            ; as an index for hrs., mins., secs., tenths
C068  B9 84 C0  RDLOOP   LDA   ALARTM,Y      ; read in alarm time or clock time to set
C06B  9D 08 DD           STA   TODTN2,X      ; store to clock—hrs. first
C06E  C8                 INY                 ; for next data position (in ALARMT or
                                             ; TIMSET)
C06F  CA                 DEX                 ; for next clock position (min., sec., tenths)
C070  10 F6              BPL   RDLOOP        ; read four bytes
C072  60                 RTS
                                             ;
                                             ; Clear the SID chip
C073  A9 00     SIDCLR   LDA   #0            ; fill with zeros
C075  A0 18              LDY   #24           ; as the offset from FRELO1
C077  99 00 D4  SIDLOP   STA   FRELO1,Y      ; store zero in each SID chip address
C07A  88                 DEY                 ; for next lower address
C07B  10 FA              BPL   SIDLOP        ; fill 25 bytes
C07D  60                 RTS                 ; we're done
                                             ;
                                             ; Restore Kernal vectors to default values.
C07E  78        RSTVEC   SEI                 ; disable IRQ interrupts while resetting IRQ
                                             ; vector
C07F  20 8A EF           JSR   RESTOR        ; reset page 3 RAM vectors to ROM table
                                             ; values
C082  58                 CLI                 ; reenable IRQ interrupts
C083  60                 RTS
                                             ;
C084  84 05 13  ALARTM   .BYTE $84,$05,$13,$0 ; hr., min., sec., tenths for alarm time
                                             ; Alarm is set for 04.05.13.0 p.m.
C088  84 05 10  TIMSET   .BYTE $84,$05,$10,$0 ; hr., min., sec., tenths for time
                                             ; Time is set for 04.05.10.0 p.m.
                                             ; For a.m., subtract $80 from hrs. place.
```

See also INTCLK, TOD1DL, TOD1RD, TOD2PR, TOD2ST.

Name

Alphabetize by swapping pointers

Description

The main alphabetizing routine does two things. First, it sets up a series of pointers to strings in memory. Then it goes through the pointers and performs a Shell sort, leaving the strings where they are, but swapping the pointers as necessary. A Shell sort is generally faster than the bubble sort used in the **ALSWAP** routine, but it's easier to write either if the fields to be sorted are the same size (which they are not in the example) or if pointers are used instead of an actual swap of strings. (Incidentally, Shell is capitalized because it's named after its inventor, Donald Shell.)

Prototype

First, create the table of pointers:

1. Look, character by character, through the zero-terminated strings.
2. When a zero is found, store the address (plus one) of the location.
3. Check the next character. If it's not zero, increment the TOTL variable and continue the loop.

Next, alphabetize the strings:

4. Set a gap variable (TOTL) initially to the number of words.
5. Clear the FLIP variable.
6. Cut the gap in half. If there are 120 words, the gap starts at 60.
7. Set a pointer (ZP) to the beginning of the list of pointers.
8. Set a second (ZQ) to the beginning of the list plus the gap.
9. Load the string pointer from ZP and store it in AP.
10. Load the second string pointer from ZQ and store in AQ.
11. Using .Y as an offset, compare the strings in AP and AQ.
12. If they're in order, skip step 13.
13. If they're not in order, swap the pointers in memory and set FLIP to a nonzero value.
14. Increment both ZP and ZQ until ZQ points beyond the end of the list.
15. If a swap has occurred, FLIP is not zero, so loop back to step 7.
16. If it has not, go back to step 6 while the gap is larger than zero.

Explanation

This is a long routine, but a good chunk of it is devoted to the part that reads a file into memory from disk. The main routine consists of three JSRs. The first calls the section that reads a text file into memory, searching for spaces—or CHR$(13)s—and replacing them with zeros as the file is copied to memory. The second calls the alphabetizing routine. The third prints out the word list.

ALPNTR itself has two primary subroutines: MAKETL and ALPHAB. The first sets up the table of pointers at $5000–$5FFF, 4096 bytes. Since each pointer needs 2 bytes, this is enough memory to handle 2048 strings or words. Note that BUFFER holds the actual words, while POINTR holds a series of pointers to the words in BUFFER.

Based on the assumption that there's at least one word in the list, the first entry in the table is set to point to the start of the buffer. Next, MAKETL searches forward for zeros. When one is found, the next address in the buffer is saved in POINTR. Each word ends with a zero byte, and the buffer itself ends with an additional zero. When the final zero is found, the loop ends.

ALPHAB is the main alphabetizing routine, and it requires several passes. Remember, the words stay where they are; it's just the pointers that are being shuffled around.

The idea of the gap is the key to the Shell sort. The gap starts out at half the number of total items in the list. If there are 56 things to put in order, the gap is 28. Entry 1 is compared with entry 29, 2 is compared with 30, and so on. If any two items are out of order, they're switched.

After the first pass, the FLIP variable is checked. If any two items have been changed, the gap's value remains the same, and the loop is repeated. If no swaps have occurred, the gap is cut in half (from 28 to 14, for example). When the gap drops to a value less than 1, the sort is finished.

The great advantage to using a gap is that it moves items quickly over a long distance. Imagine that *zookeeper* is the first word on a list of, say, 500 words, and that its rightful place in the alphabetized list is last. On the first pass (gap of 250), it is moved 250 places, from 1 to 251. On the next pass (gap of 125), it jumps another 125. After just two comparisons, it has traveled from location 1 to location 376. In an ordinary bubble sort, it would take 375 comparisons—375 passes through the

loop—to move that far. A Shell sort of a medium-sized list will almost always beat a bubble sort.

The following program is written in reasonably short modules and should be easy to follow. One technique worth noting occurs at $C069, where DBLINC calls the routine INCZPZQ directly below it. The INCZPZQ routine adds 1 to the pointers at ZP and ZQ. Because the DBLINC (double increment) routine is placed above the routine that increments once, the routine is called twice. The end RTS first returns to just past DBLINC, where the routine executes a second time, after which the RTS returns to the place that called it.

Routine

```
C000                ZP       =    $FB
C000                ZQ       =    $FD
C000                AP       =    $F7          ; for the 128, use other available zero-page
                                              ; locations here
C000                AQ       =    $F9          ; and here
C000                STATUS   =    144
C000                CHROUT   =    $FFD2
C000                BUFFER   =    $6000        ; storage area where the words will be loaded
                                              ; into memory
C000                POINTR   =    $5000        ; table of two-byte pointers to the words
                                              ; (maximum 2048 from $5000 through $5FFF)
                                              ; LDA #0, set for bank 15 (128 only)
                                              ; STA $FF00, (128 only)
C000  20  17  C1    MAIN     JSR  READFILE     ; read a file from disk
                                              ; LDA #63, set for bank 0 (128 only)
                                              ; STA $FF00, (128 only)
C003  20  0A  C0             JSR  ALPNTR       ; alphabetize the word list
                                              ; LDA #0, set for bank 15 (128 only)
                                              ; STA $FF00; (128 only)
C006  20  92  C1             JSR  PRINTM       ; print it out
C009  60                     RTS
                                              ;
C00A                ALPNTR   =    *            ; alphabetize by pointers
C00A  20  11  C0             JSR  MAKETL       ; make a table of pointers
C00D  20  8F  C0             JSR  ALPHAB       ; alphabetize it
C010  60                     RTS
                                              ;
C011                MAKETL   =    *            ; create the table
                                              ; Set things up.
C011  A9  [illegible]          LDA  #147         ; clear screen character
C013  20  D2  FF             JSR  CHROUT       ; print it
C016  20  58  C0             JSR  SETZPAP      ; point ZP to POINTR and AP to BUFFER
C019  A0  00                 LDY  #0
C01B  8C  12  C1             STY  TOTL         ; zero the counter
C01E  8C  13  C1             STY  TOTL+1
                                              ;
C021  A5  F7        BIGLOP   LDA  AP           ; low byte of pointer to BUFFER
C023  91  FB                 STA  (ZP),Y       ; store it in the table
C025  20  6C  C0             JSR  INCZPZQ      ; increment ZP and ZQ
C028  A5  [illegible]          LDA  AP+1         ; high byte
C02A  91  FB                 STA  (ZP),Y       ; store it
C02C  20  6C  C0             JSR  INCZPZQ      ; and ZP/ZQ go up
C02F  20  B6  C0             JSR  PLUSTL       ; increment the counter
                                              ;
C032  B1  F7                 LDA  (AP),Y       ; check the first byte
```

```
C034  D0 10          BNE   MORE        ; if not a zero, there are more words
C036  A9 0D          LDA   #13         ; print a RETURN
C038  20 D2 FF       JSR   CHROUT
C03B  A5 F7          LDA   AP          ; save the last pointer
C03D  8D 15 C1       STA   BUFEND      ; into BUFEND
C040  A5 F8          LDA   AP+1        , high byte
C042  8D 16 C1       STA   BUFEND+1
C045  60             RTS               ; main RTS of MAKETL routine
C046  A9 2A  MORE    LDA   #42         ; take an asterisk
C048  20 D2 FF       JSR   CHROUT      ; print it
C04B  20 79 C0 SMALLP JSR  INCAPAQ     ; increment AP and AQ
C04E  B1 F7          LDA   (AP),Y      ; check the next one
C050  D0 F9          BNE   SMALLP      ; go back if not zero
C052  20 79 C0       JSR   INCAPAQ     ; INC the pointer (to the start of next word)
C055  4C 21 C0       JMP   BIGLOP      ; and go back
                                       ;
C058  A9 00  SETZPAP LDA   #<BUFFER    ; put the address of buffer
C05A  85 F7          STA   AP          ; into AP
C05C  A9 60          LDA   #>BUFFER
C05E  85 F8          STA   AP+1
C060  A9 00          LDA   #<POINTR    ; and the address of POINTR
C062  85 FB          STA   ZP          ; into ZP
C064  A9 50          LDA   #>POINTR
C066  85 FC          STA   ZP+1
C068  60             RTS
                                       ;
C069  20 6C C0 DBLINC JSR  INCZPZQ     ; call it once and then fall through for
                                       ; double INC
C06C  E6 FB  INCZPZQ INC   ZP          ; ZP points higher
C06E  D0 02          BNE   IPQ1
C070  E6 FC          INC   ZP+1        ; handle the high byte
C072  E6 FD  IPQ1    INC   ZQ          ; ZQ, too
C074  D0 02          BNE   IPQ2
C076  E6 FE          INC   ZQ+1        ; high byte
C078  60     IPQ2    RTS               ; that's all, folks
                                       ;
C079  E6 F7  INCAPAQ INC   AP          ; AP points higher
C07B  D0 02          BNE   IAQ1
C07D  E6 F8          INC   AP+1        ; if AP = 0, INC the high byte
C07F  E6 F9  IAQ1    INC   AQ          ; AQ goes up by 1
C081  D0 02          BNE   IAQ2
C083  E6 FA          INC   AQ+1        ; and maybe the high byte
C085  60     IAQ2    RTS               ; all done
                                       ;
C086  EE 12 C1 PLUSTL INC  TOTL        ; add 1 to the total
C089  D0 03          BNE   PLT1
C08B  EE 13 C1       INC   TOTL+1      ; high byte, too
C08E  60     PLT1    RTS
                                       ;
                                       ; The main alphabetizing routine.
                                       ;
C08F         ALPHAB  =     *
C08F  20 A0 C0 ALPLOP JSR  INITPQ      ; set up the initial pointers in ZP and ZQ
C092  20 BA C0       JSR   SHUFFLE     ; move them around and put them in order
C095  AD 14 C1       LDA   FLIP        ; if the flag is set,
C098  D0 F5          BNE   ALPLOP      ; go back and do it again
C09A  20 FD C0       JSR   HFTOTL      ; cut TOTL in half
C09D  B0 F0          BCS   ALPLOP      ; if carry set, do more
C09F  60             RTS               ; otherwise, we're done
                                       ;
C0A0  A0 00  INITPQ  LDY   #0          ;
C0A2  8C 14 C1       STY   FLIP        ; reset the FLIP flag
C0A5  20 58 C0       JSR   SETZPAP     ; set ZP to POINTR address
C0A8  AD 12 C1       LDA   TOTL        ;
```

```
C0AB 29  FE               AND  #%11111110   ; round down to nearest even number
C0AD 18                   CLC
C0AE 65  FB               ADC  ZP           ; add in low byte
C0B0 85  FD               STA  ZQ           ; higher pointer in ZQ
C0B2 AD  13  C1           LDA  TOTL+1       ; add the high byte
C0B5 65  FC               ADC  ZP+1         ; to ZP+1
C0B7 85  FE               STA  ZQ+1         ; and put it in ZQ
C0B9 60                   RTS               ; end of INITPQ
                                            ;
C0BA             SHUFFLE  =    *
C0BA A0  00               LDY  #0
C0BC B1  FB               LDA  (ZP),Y       ; get the first pointer
C0BE 85  F7               STA  AP           ; and set up a pointer
C0C0 B1  FD               LDA  (ZQ),Y       ; and the second
C0C2 85  F9               STA  AQ           ; as well
C0C4 C8                   INY               ; now the high bytes
C0C5 B1  FB               LDA  (ZP),Y
C0C7 85  F8               STA  AP+1
C0C9 B1  FD               LDA  (ZQ),Y
C0CB 85  FA               STA  AQ+1
                                            ;
C0CD 88                   DEY               ; back to zero
C0CE B1  F9               LDA  (AQ),Y       ; look for the zero at the end of the table
C0D0 D0  01               BNE  KEEPON       ; if the first character of (AQ) isn't zero, we
                                            ; have more
C0D2 60                   RTS               ; else, finish this routine
C0D3 B1  F7      KEEPON   LDA  (AP),Y
C0D5 F0  20               BEQ  NOSWIT       ; found a zero at the end of the (shorter)
                                            ; string from AP
C0D7 D1  F9               CMP  (AQ),Y       ; not a zero, so compare to the AQ string
C0D9 90  1C               BCC  NOSWIT       ; if AP < AQ, no switch
C0DB D0  04               BNE  SWITCH       ; if not equal, AQ < AP
C0DD C8                   INY               ; else they're equal and we check some
                                            ; more
C0DE 4C  D3  C0           JMP  KEEPON
                                            ;
C0E1 8D  14  C1  SWITCH   STA  FLIP         ; store a nonzero value in FLIP
C0E4 A0  00               LDY  #0
C0E6 A5  F7               LDA  AP           ; get the pointer from AP
C0E8 91  FD               STA  (ZQ),Y       ; and put it in the table
C0EA A5  F9               LDA  AQ           ; same for AQ
C0EC 91  FB               STA  (ZP),Y       ; low byte
C0EE C8                   INY
C0EF A5  F8               LDA  AP+1         ; now the high bytes
C0F1 91  FD               STA  (ZQ),Y
C0F3 A5  FA               LDA  AQ+1
C0F5 91  FB               STA  (ZP),Y       ; and fall through
                                            ;
C0F7 20  69  C0  NOSWIT   JSR  DBLINC       ; double increment of ZP and ZQ
C0FA 4C  BA  C0           JMP  SHUFFLE
                                            ; end of SHUFFLE
                                            ;
C0FD 4E  13  C1  HFTOTL   LSR  TOTL+1       ; shift right (cut in half) the high byte of
                                            ; TOTL
C100 6E  12  C1           ROR  TOTL         ; and the low byte
C103 38                   SEC               ; set carry means more
C104 AD  13  C1           LDA  TOTL+1       ; is there a high byte?
C107 D0  08               BNE  ENDHF        ; yes, there's more
C109 AD  12  C1           LDA  TOTL         ; no, check the low byte
C10C C9  02               CMP  #2           ; if it's 2 or more
C10E B0  01               BCS  ENDHF        ; we're OK
C110 18                   CLC               ; else clear carry (all done)
```

```
C111  60         ENDHF     RTS              ; as we leave, CLC means done, SEC means
                                            ; keep going
                                            ;
C112  00  00     TOTL      .BYTE  0,0
C114  00         FLIP      .BYTE  0
C115  00  00     BUFEND    .BYTE  0,0
                                            ;
C117             READFILE  =      *
C117             SETLFS    =      65466
C117             SETNAM    =      65469
C117             OPEN      =      65472
C117             CHKIN     =      65478
C117             CHRIN     =      65487
C117             CLOSE     =      65475
C117             CLRCHN    =      65484
                                            ;
C117  A9  01               LDA    #1        ; logical file number
C119  A2  08               LDX    #8        ; device number for disk drive
C11B  A0  02               LDY    #2        ; secondary address (2-14 are OK)
C11D  20  BA  FF           JSR    SETLFS
C120  A9  0D               LDA    #FNLEN    ; length of filename
C122  A2  85               LDX    #<FNAME   ; address of filename
C124  A0  C1               LDY    #>FNAME
C126  20  BD  FF           JSR    SETNAM
C129  20  C0  FF           JSR    OPEN
C12C  A2  01               LDX    #1        ; logical file number
C12E  20  C6  FF           JSR    CHKIN     ; set for input
                                            ;
C131  A9  00               LDA    #<BUFFER  ; set up a pointer
C133  85  FB               STA    ZP
C135  A9  60               LDA    #>BUFFER  ; high byte
C137  85  FC               STA    ZP+1
                                            ;
C139  A0  00               LDY    #0
C13B  20  CF  FF GETCHR    JSR    CHRIN     ; get a character
C13E  C9  0D               CMP    #13       ; check for RETURN
C140  F0  26               BEQ    DELIMIT
C142  C9  20               CMP    #32       ; look for a space
C144  90  09               BCC    CHKEND    ; eliminate characters 0-31
C146  F0  20               BEQ    DELIMIT   ; spaces are delimiters
C148  91  FB               STA    (ZP),Y
C14A  C8                   INY
C14B  D0  02               BNE    CHKEND    ; check for the end
C14D  E6  FC               INC    ZP+1      ; increment the pointer
C14F  A6  90     CHKEND    LDX    STATUS
C151  F0  E8               BEQ    GETCHR    ; if equal, get more characters
C153  A9  00               LDA    #0        ; close it up with three zeros
C155  91  FB               STA    (ZP),Y    ; store it
C157  20  76  C1           JSR    ADDYZP    ; reset ZP
C15A  91  FB               STA    (ZP),Y
C15C  C8                   INY
C15D  91  FB               STA    (ZP),Y
C15F  A9  01               LDA    #1
C161  20  C3  FF           JSR    CLOSE     ; close the file
C164  20  CC  FF           JSR    CLRCHN    ; clear channels
C167  60                   RTS              ; the end of the routine
                                            ;
C168  C0  00     DELIMIT   CPY    #0        ; is this the first character?
C16A  F0  E3               BEQ    CHKEND    ; yes, go back
                                            ;
                                            ; Enter this routine if a space or RETURN is
                                            ; found after a word.
C16C  A9  00               LDA    #0        ; zero marks the division
C16E  91  FB               STA    (ZP),Y    ; put a zero in memory
```

```
C170  20 76 C1          JSR   ADDYZP       ; add Y to ZP (plus 1)
C173  4C 4F C1          JMP   CHKEND       ; and check for end of file
                                           ;
C176  38        ADDYZP  SEC                ; add 1 to .Y
C177  98                TYA                ; put it in .A
C178  65 FB             ADC   ZP           ; add to ZP
C17A  85 FB             STA   ZP           ; fix ZP
C17C  A9 00             LDA   #0           ; handle the high byte
C17E  A8                TAY                ; put zero back into .Y
C17F  65 FC             ADC   ZP+1         ; add
C181  85 FC             STA   ZP+1         ; and store
C183  98                TYA                ; exit with zero in .A
C184  60                RTS
                                           ;
C185  41 53 43  FNAME   ASC   "0:ASCIIFILE,S,R"
                                           ; name of file to read
C192            FNLEN   =     *    FNAME
                                           ;
C192  20 58 C0  PRINTM  JSR   SETZPAP      ; set ZP to point to POINTR table
C195  A0 01     PMLOOP  LDY   #1
C197  B1 FB             LDA   (ZP),Y       ; get the POINTR high byte
C199  85 F8             STA   AP+1         ; set up AP
C19B  88                DEY
C19C  B1 FB             LDA   (ZP),Y       ; now the low byte
C19E  85 F7             STA   AP           ; (and .Y holds a zero)
C1A0  B1 F7             LDA   (AP),Y       ; is the first character a zero?
C1A2  F0 15             BEQ   QUITIT       ; if so, we're all done
C1A4  20 D2 FF  PINLOP  JSR   CHROUT       ; no, print it
C1A7  20 79 C0          JSR   INCAPAQ      ; AP increases by 1
C1AA  B1 F7             LDA   (AP),Y       ; get the next character
C1AC  D0 F6             BNE   PINLOP       ; until there's a zero
C1AE  A9 0D             LDA   #13          ; print RETURN
C1B0  20 D2 FF          JSR   CHROUT
C1B3  20 69 C0          JSR   DBLINC       ; move ZP up two notches
C1B6  4C 95 C1          JMP   PMLOOP       ; and set up the next address
                                           ;
C1B9  60        QUITIT  RTS
```

See also ALSWAP, SRCBIN.

Name

Alphabetize a list by swapping strings that are out of order

Description

Although the example program is longer than most others in this book, it's short for an alphabetizing routine. (See **ALPNTR** for a longer, but much faster routine.) For reasons explained below, **ALSWAP** uses a relatively slow bubble-sort algorithm, which at machine language speeds is fast enough if the list to be sorted has either fixed-length records or a small-to-medium number of variable-length records.

Prototype

1. Count the number of records. Each word is a record in the example program.
2. Start by setting two zero-page pointers: one pointer (ZP) to the first record and another (ZQ) to the second.
3. Decrement the counter for number of records. If it's zero, exit.
4. Otherwise, copy the counter to a second variable (INCOUNT).
5. Compare the two records.
6. If they're out of place, swap them.
7. Continue the inner loop by decrementing INCOUNT and incrementing the pointers to the two records. Branch back to step 5.
8. When the inner loop counter INCOUNT reaches zero, branch to step 3.

Explanation

The strings in the example program were selected randomly from a book of folktales. Each is terminated by a zero byte. The three primary subroutines in the framing routine are COUNTEM, **ALSWAP**, and PRINTEM.

COUNTEM cruises through memory, finding the zero terminators and generally counting the number of words in the list. When the number of words is known, **ALSWAP** alphabetizes them.

Two zero-page pointers hold the addresses of two neighboring strings. Start by comparing the first to the second. Then compare the second to the third, and so on.

The COMPAR subroutine ($C08F–$C0AA) makes a decision about the two strings' positions. If they're in the right order, the carry flag is cleared and the subroutine ends. If not, carry is set. Back in the main alphabetizing routine, a BCC

skips ahead if the words are in their proper places. Otherwise, the strings switch positions.

The SWITCH routine handles the trading of two strings. If the two strings are next to each other, it's relatively easy to make them trade places; "THERE@HI@" takes up the same amount of memory as "HI@THERE@" (the @s represent the zero terminators). If the two strings "THERE" and "HI" occupy different parts of the list, a variety of time-wasting memory moves are necessary just to get the words in the right places.

After the comparison (COMPAR) and the trade (SWITCH), we check the next two strings, until the inner loop has finished. The outer loop counts backward to 1 (from one less than the number of items on the list).

The alphabetizing routine ends, and the PRINTEM routine takes over, listing the words in order.

Routine

```
C000                ZP        =    $FB
C000                ZQ        =    $FD
C000                CHROUT    =    $FFD2
                                                  ;
C000  20 0A C0                JSR  COUNTEM        ; count the number of words
C003  20 32 C0                JSR  ALSWAP         ; alphabetize by swapping
C006  20 EB C0                JSR  PRINTEM        ; print them in order
C009  60                      RTS                 ; end of this routine
                                                  ;
C00A  A0 00         COUNTEM   LDY  #0             ; first zero out the counter
C00C  8C 1B C1                STY  COUNTER        ; low byte
C00F  8C 1C C1                STY  COUNTER+1      ; high byte
C012  20 0E C1                JSR  BUF2ZP         ; copy the address of buffer to ZP
C015  B1 FB         CNLOOP    LDA  (ZP),Y         ; get a first character
C017  D0 01                   BNE  CNMORE         ; if it's not zero, continue
C019  60                      RTS                 ; done
C01A  EE 1B C1      CNMORE    INC  COUNTER        ; counter up one
C01D  D0 03                   BNE  FIND0
C01F  EE 1C C1                INC  COUNTER+1      ; high-byte increments
C022  C8            FIND0     INY                 ; increase the .Y counter
C023  D0 02                   BNE  LOOKMORE       ; if Y <> 0, continue
C025  E6 FC                   INC  ZP+1           ; else, add 256 to ZP
C027  B1 FB         LOOKMORE  LDA  (ZP),Y         ; get the next character
C029  D0 F7                   BNE  FIND0          ; if not zero, keep going
                                                  ;
C02B  C8                      INY                 ; keep the index going
C02C  D0 E7                   BNE  CNLOOP         ; go back for more
C02E  E6 FC                   INC  ZP+1           ; handle ZP if Y = 0
C030  D0 E3                   BNE  CNLOOP         ; branch always
                                                  ;
                                                  ; ALSWAP—the main routine for
                                                  ; alphabetizing.
C032                ALSWAP    =    *
C032  20 0E C1      ALUTLP    JSR  BUF2ZP         ; set up ZP and ZQ pointers
C035  20 58 C0                JSR  CNDOWN         ; counter down by one
C038  20 77 C0      ALINLP    JSR  ZPZQ           ; copy ZP to ZQ
C03B  20 80 C0                JSR  FINWORD        ; find the next word for ZQ
C03E  20 8F C0                JSR  COMPAR         ; compare the two words
C041  90 03                   BCC  SKIP           ; if CC, leave them alone
```

```
C043  20  AB  C0             JSR   SWITCH       ; else, switch them
C046  CE  1D  C1  SKIP       DEC   INCOUNT      ; are we done?
C049  D0  ED                 BNE   ALINLP       ; no, continue the inner loop
C04B  CE  1E  C1             DEC   INCOUNT+1    ; else, INCOUNT = 0, so check the high
                                                ; byte
C04E  AD  1E  C1             LDA   INCOUNT+1
C051  C9  FF                 CMP   #255         ; 255 means we're done
C053  D0  E3                 BNE   ALINLP       ; if not 255, continue
C055  4C  32  C0             JMP   ALUTLP       ; go back for outer loop
                                                ;
C058  CE  1B  C1  CNDOWN     DEC   COUNTER      ; counter down by one
C05B  D0  0D                 BNE   COPYUI       ; if not zero, we're OK
C05D  CE  1C  C1             DEC   COUNTER+1    ; DEC the high byte
C060  AD  1C  C1             LDA   COUNTER+1    ; check it
C063  C9  FF                 CMP   #255         ; if 255, we're all done
C065  D0  03                 BNE   COPYUI       ; if not, continue
C067  68                     PLA
C068  68                     PLA                ; trash the return address
C069  60                     RTS                ; return to the previous routine
C06A  AD  1B  C1  COPYUI     LDA   COUNTER
C06D  8D  1D  C1             STA   INCOUNT
C070  AD  1C  C1             LDA   COUNTER+1
C073  8D  1E  C1             STA   INCOUNT+1    ; copy COUNTER to INCOUNT
C076  60                     RTS
                                                ;
C077  A5  FB      ZPZQ       LDA   ZP           ; copy ZP
C079  85  FD                 STA   ZQ           ; to ZQ
C07B  A5  FC                 LDA   ZP+1         ; and the high byte
C07D  85  FE                 STA   ZQ+1         ; as well
C07F  60                     RTS
                                                ;
C080              FINWORD    =     *            ; finds the next word
C080  A0  00                 LDY   #0           ; index for ZQ
C082  B1  FD      FINLP      LDA   (ZQ),Y       ; get a character
C084  E6  FD                 INC   ZQ           ; the counter must go forward
C086  D0  02                 BNE   CHECKQ
C088  E6  FE                 INC   ZQ+1         ; handle the high byte
C08A  C9  00      CHECKQ     CMP   #0           ; check the character we get
C08C  D0  F4                 BNE   FINLP        ; if it isn't zero, go back
C08E  60                     RTS                ; but if it is, we're done
                                                ;
C08F  A0  00      COMPAR     LDY   #0
C091  B1  FB      COMLP      LDA   (ZP),Y       ; get a character from the first word
C093  F0  0A                 BEQ   RIGHT        ; the first is shorter, so quit
C095  D1  FD                 CMP   (ZQ),Y       ; compare it
C097  90  06                 BCC   RIGHT        ; if ZP < ZQ, they're right
C099  D0  0E                 BNE   WRONG        ; if not equal, they're in the wrong order
C09B  C8                     INY                ; else try for more
C09C  4C  91  C0             JMP   COMLP
                                                ;
C09F  A5  FD      RIGHT      LDA   ZQ           ; set up ZP for the next word
C0A1  85  FB                 STA   ZP           ; copy low byte
C0A3  A5  FE                 LDA   ZQ+1         ; and also
C0A5  85  FC                 STA   ZP+1         ; the high
C0A7  18                     CLC                ; a flag that means it's OK
C0A8  60                     RTS                ; and we're done here
                                                ;
C0A9  38          WRONG      SEC                ; carry set = a problem
C0AA  60                     RTS                ; now SWITCH will be called
                                                ;
C0AB  A0  00      SWITCH     LDY   #0
C0AD  38                     SEC                ; carry should be set, but we'll make sure
C0AE  B1  FB      SWILP      LDA   (ZP),Y
C0B0  D0  01                 BNE   AHEAD        ; if it's not zero
```

```
C0B2  18                 CLC              ; if we get a zero, clear carry to mark the
                                          ; end of the first word
C0B3  99 7D C1  AHEAD    STA  TEMBUF,Y    ; save the word from ZP
C0B6  B1 FD              LDA  (ZQ),Y      ; copy ZQ
C0B8  91 FB              STA  (ZP),Y      ; to ZP
C0BA  F0 03              BEQ  SWI2        ; the end of ZQ is a zero
C0BC  C8                 INY              ; otherwise, keep going
C0BD  D0 EF              BNE  SWILP       ; branch back (always)
C0BF  90 11     SWI2     BCC  COPY2       ; if carry clear, the first word was shorter
C0C1  8C 1F C1           STY  TEMPY       ; stash Y
C0C4  C8                 INY
C0C5  B1 FB     LOOP2    LDA  (ZP),Y      ; get more characters
C0C7  99 7D C1           STA  TEMBUF,Y
C0CA  F0 03              BEQ  LOTY
C0CC  C8                 INY
C0CD  D0 F6              BNE  LOOP2       ; if not, keep going
C0CF  AC 1F C1  LOTY     LDY  TEMPY       ; get Y back
C0D2  C8        COPY2    INY              ; INC .Y to point one past the current (zero)
                                          ; byte
C0D3  98                 TYA              ; put it in .A
C0D4  18                 CLC
C0D5  65 FB              ADC  ZP          ; add it to ZP
C0D7  85 FB              STA  ZP          ; and store it
C0D9  A9 00              LDA  #0          ; do the high byte, too
C0DB  A8                 TAY              ; set up Y for the next loop
C0DC  65 FC              ADC  ZP+1        ; add zero plus carry
C0DE  85 FC              STA  ZP+1        ; store it
                                          ; Now ZP points to a new location.
C0E0  B9 7D C1  LASTLP   LDA  TEMBUF,Y
C0E3  91 FB              STA  (ZP),Y
C0E5  D0 01              BNE  INNNY       ; if not zero, continue
C0E7  60                 RTS              ; we're done
C0E8  C8        INNNY    INY              ; or we're not
C0E9  D0 F5              BNE  LASTLP      ; and loop back
                                          ;
C0EB  20 0E C1  PRINTEM  JSR  BUF2ZP
C0EE  A0 00              LDY  #0
C0F0  B1 FB     FIRST    LDA  (ZP),Y      ; get the first character
C0F2  D0 01              BNE  NOTDONE
C0F4  60                 RTS              ; if it's a zero, finish this routine
C0F5  20 D2 FF  NOTDONE  JSR  CHROUT
C0F8  C8                 INY
C0F9  D0 02              BNE  NOTEQ
C0FB  E6 FC              INC  ZP+1        ; take care of the high byte
C0FD  B1 FB     NOTEQ    LDA  (ZP),Y      ; get more characters
C0FF  D0 F4              BNE  NOTDONE
C101  A9 0D              LDA  #13         ; print a return
C103  20 D2 FF           JSR  CHROUT
C106  C8                 INY
C107  D0 02              BNE  ZIZL
C109  E6 FC              INC  ZP+1
C10B  4C F0 C0  ZIZL     JMP  FIRST
                                          ;
C10E  A9 20     BUF2ZP   LDA  #<BUFFER    ; set up a pointer to BUFFER
C110  85 FB              STA  ZP          ; low byte of BUFFER to ZP
C112  85 FD              STA  ZQ          ; also in ZQ
C114  A9 C1              LDA  #>BUFFER
C116  85 FC              STA  ZP+1        ; high byte to ZP
C118  85 FE              STA  ZQ+1        ; ZQ, too
C11A  60                 RTS
                                          ;
C11B  00 00     COUNTER  .BYTE 0,0
C11D  00 00     INCOUNT  .BYTE 0,0
C11F  00        TEMPY    .BYTE 0
```

```
                                                ;
C120  41  4E  44  BUFFER   .ASC   "AND"
C123  00                   .BYTE  0
C124  43  4C  45           .ASC   "CLEAR"
C129  00                   .BYTE  0
C12A  53  54  55           .ASC   "STUMPS"
C130  00                   .BYTE  0
C131  57  45               .ASC   "WE"
C133  00                   .BYTE  0
C134  46  4F  4C           .ASC   "FOLKS"
C139  00                   .BYTE  0
C13A  54  48  45           .ASC   "THEY"
C13E  00                   .BYTE  0
C13F  54  48  45           .ASC   "THEN"
C143  00                   .BYTE  0
C144  52  45  4D           .ASC   "REMEMBER"
C14C  00                   .BYTE  0
C14D  59  4F  55           .ASC   "YOU"
C150  00                   .BYTE  0
C151  53  45  45           .ASC   "SEEN"
C155  00                   .BYTE  0
C156  54  4F               .ASC   "TO"
C158  00                   .BYTE  0
C159  54  57  45           .ASC   "TWENTY"
C15F  00                   .BYTE  0
C160  47  45  4E           .ASC   "GENERALLY"
C169  00                   .BYTE  0
C16A  44  4F  47           .ASC   "DOG"
C16D  00                   .BYTE  0
C16E  41  42  4F           .ASC   "ABOUT"
C173  00                   .BYTE  0
C174  53  54  52           .ASC   "STRIPE"
C17A  00                   .BYTE  0
C17B  00  00               .BYTE  0,0
C17D              TEMBUF   =      *             ; temporary buffer (allow 256 bytes)
```

See also ALPNTR, SRCBIN.

Name

Animation: alternating character sets

Description

This is one of the easier ways to animate characters on the 40-column screen of the 64 or 128. If you press SHIFT and the Commodore key at the same time, the character set will switch between uppercase/graphics mode and lowercase/uppercase mode. By alternately printing CHR$(14) and CHR$(142), you can cause any or all of the characters on the screen to change.

Prototype

1. Check a timer (the jiffy clock, in this example).
2. If enough time has passed, start at step 3 below. Otherwise, exit the routine.
3. Add a constant to the timer and store it for the next time.
4. Load .A with the FLIP value, which is either 14 or 142. Print it and then, using EOR, change it to the other value.
5. Move characters that should be in motion.

Explanation

Although the example provides a lively screen, there's really no movement of characters at all. The alternating *M*'s and *W*'s are the effect we're looking for—animation via character set flipping. The character that flips between C and a dash is another by-product of this technique. It seems to move from left to right, but it's not actually being placed and erased. The line where it moves contains a series of 40 C characters, but at any given point, 39 of them are black, which is the background color. The apparent motion comes from a different value being stored into color memory.

There aren't a lot of interesting dual characters in the two built-in character sets, but if you define your own custom characters, you can achieve some very interesting effects.

Routine

```
C000                JIF      =    $A2          ; LSB of the jiffy clock
C000                COLMEM   =    55296        ; color memory
C000                SCRMEM   =    1024         ; screen memory
C000                LINCOL   =    COLMEM+80
C000                LINSCR   =    SCRMEM+80
C000                BKGRND   =    53281        ; background register
C000                CHROUT   =    $FFD2        ; Kernal routines
C000                GETIN    =    $FFE4
C000  A9 00                  LDA  #0
C002  8D 21 D0               STA  BKGRND       ; background color = black
C005  A9 05                  LDA  #5           ; ASCII code for white
C007  20 D2 FF               JSR  CHROUT       ; print it
C00A  A9 93                  LDA  #147         ; ASCII for clear screen
```

```
C00C  20  D2  FF           JSR   CHROUT      ; print it, also
C00F  A0  00               LDY   #0
C011  B9  71  C0  PRLOOP   LDA   STRING,Y
C014  F0  06               BEQ   PROUT       ; if zero, quit
C016  20  D2  FF           JSR   CHROUT      ; print it
C019  C8                   INY               ; count up
C01A  D0  F5               BNE   PRLOOP      ; branch back
C01C  20  28  C0  PROUT    JSR   SETUP       ; set up the animation characters
C01F  20  3D  C0  CONT     JSR   ANIMAT      ; animate
C022  20  E4  FF           JSR   GETIN       ; get a key
C025  F0  F8               BEQ   CONT        ; if no key, continue
C027  60                   RTS
                                             ;
C028  A0  00      SETUP    LDY   #0
C02A  8C  70  C0           STY   POSITION    ; start at zero
C02D  A9  00      SETLOP   LDA   #0          ; color for black
C02F  99  50  D8           STA   LINCOL,Y    ; store in color memory
C032  A9  43               LDA   #67         ; shifted C screen code
C034  99  50  04           STA   LINSCR,Y
C037  C8                   INY               ; count forward
C038  C0  28               CPY   #40         ; to 39
C03A  D0  F1               BNE   SETLOP      ; loop back
C03C  60                   RTS
                                             ;
C03D              ANIMAT   =     *
C03D  AD  92  C0           LDA   TIMER       ; check the timer
C040  C5  A2               CMP   JIF         ; is it time yet?
C042  F0  01               BEQ   MOVEM       ; yes, move ahead
C044  60                   RTS               ; otherwise, go back
                                             ;
C045  18          MOVEM    CLC               ; .A already holds the current jiffy value
C046  69  0A               ADC   #10         ; add ten jiffies (1/6 second)
C048  8D  92  C0           STA   TIMER       ; remember it
C04B  AD  93  C0           LDA   FLIP        ; either 14 or 142
C04E  20  D2  FF           JSR   CHROUT      ; print it
C051  49  80               EOR   #$80        ; change it to the other one (14 or 142)
C053  8D  93  C0           STA   FLIP        ; and save it
C056  10  00               BPL   AHEAD       ; if it's 14, move ahead
                                             ; RTS; else, quit (optional)
                                             ;
C058  AC  70  C0  AHEAD    LDY   POSITION    ; where is the character?
C05B  A9  00               LDA   #0          ; black
C05D  99  50  D8           STA   LINCOL,Y    ; clear it out
C060  C8                   INY               ; move ahead one space
C061  C0  28               CPY   #40         ; is it 40 yet?
C063  D0  02               BNE   WHITE       ; no
C065  A0  00               LDY   #0          ; yes, make it zero
C067  8C  70  C0  WHITE    STY   POSITION    ; remember .Y
C06A  A9  01               LDA   #1          ; the color code for white
C06C  99  50  D8           STA   LINCOL,Y    ; store it
C06F  60                   RTS
                                             ;
C070  00          POSITION .BYTE 0
C071  57  57  57  STRING   .ASC  "WWWWWWW"
C078  0D  B1  B1           .BYTE 13,177,177,177,177,177,177,177
C080  0D                   .BYTE 13
C081  0D  B2  B2           .BYTE 13,178,178,178,178,178,178,178,13
C08A  4D  4D  4D           .ASC  "MMMMMMM"
C091  00                   .BYTE 0
C092  00          TIMER    .BYTE 0
C093  0E          FLIP     .BYTE 14
```

See also CHRDEF, CUST80.

Name

Convert a signed byte value to a signed integer value

Description

This very short routine changes an 8-bit signed number into a 16-bit signed number.

Prototype

1. Copy the original byte to the low byte of the integer.
2. If the sign bit (bit 7) is set, store an $FF to the high byte.
3. If bit 7 is clear, store a $00 to the high byte.

Explanation

A memory location can hold only 256 possible numbers. In unsigned arithmetic, the numbers are 0–255. Signed arithmetic also allows 256 numbers, but they range from −128 to +127. If that sounds confusing, think of a clock with the numbers 1–12. Add one hour to 3:00, and the clock shows 4:00. But if you add ten hours to 3:00, the result is 1:00, because there are no hours beyond 12:00. In a sense, adding 10 to 3:00 is the same as subtracting 2 from 3:00, so 10 = −2 when you're using a clock. In signed arithmetic, a 255 is the same as −1, a 254 is −2, and so on. If a memory location holds a zero and you use the DECrement instruction, it will now hold an $FF, which can be called a 255 (unsigned) or a −1 (signed).

Bit 7 indicates whether a number is positive (0) or negative (1). The numbers 0–127 (%00000000–%01111111) all have a 0 in the high bit. Likewise, the numbers from −128 through −1 (%10000000–%11111111) contain a 1 in the sign bit.

Two-byte signed integer values follow the same rules, but the numbers fall between −32768 and 32767 and bit 15 is the sign bit. The number −1 is $FFFF instead of $FF, and the number +1 is $0001 instead of $01. Thus, to make a positive byte into a positive two-byte integer, we have to add a $00 as the high byte. For negative bytes, an $FF becomes the high byte.

The example routine copies the original value to the low byte of the integer. It then checks the sign bit and puts the appropriate value ($00 or $FF) into the high byte of the integer.

Routine

```
C000  AD 15  C0  B2SNIN   LDA   NUMBER      ; the byte we're copying
C003  8D 16  C0           STA   INTGER      ; into the low byte of INTGER
C006  2A                  ROL               ; check the sign bit
C007  B0 06               BCS   NEGATV      ; branch ahead if negative
C009  A9 00               LDA   #%00000000  ; it's positive
C00B  8D 17  C0           STA   INTGER+1    ; so clear the high byte
```

```
C00E  60                      RTS                   ; and we're done
                                                    ;
C00F  A9 FF         NEGATV    LDA   #%11111111      ; the number is negative
C011  8D 17  C0               STA   INTGER+1        ; so fill the high byte with ones
C014  60                      RTS                   ; and we're done
                                                    ;
C015  09            NUMBER    .BYTE 09
C016  00 00         INTGER    .BYTE 00,00
```

See also B2UNIN, BCD2BY, CB2BCD, CFP2I, CI2FP, CNVBFP.

Name

Convert a byte value (8 bits) to an unsigned integer value (16 bits)

Description

This is a very simple routine that adds a high byte of $00 to a byte value to make it an unsigned integer value.

Prototype

1. Copy the original byte to the low byte of the integer.
2. Put a zero in the high byte of the integer.

Explanation

Bytes, by their very nature, can contain only the numbers 0–255 ($00–$FF). By combining two bytes to represent a single number, you can extend the range to 0–65535 ($0000–$FFFF). On the 64 and 128, the convention is to put the low byte in front of the high byte. The number $A012, for example, would be stored in memory as 18 ($12) followed by 160 ($A0).

This routine merely copies the byte to the first position of the integer and then tacks on a zero for the high byte.

Routine

```
C000  AD 0C  C0  B2UNIN   LDA   NUMBER      ; the byte we're copying
C003  8D 0D  C0           STA   INTGER      ; into the low byte of integer
C006  A9 00               LDA   #0          ; if it's unsigned, always a zero
C008  8D 0E  C0           STA   INTGER+1    ; the high byte
C00B  60                  RTS
                                            ;
                                            ; data bytes
C00C  09             NUMBER   .BYTE 09
C00D  00  00         INTGER   .BYTE 00,00
```

See also B2SNIN, BCD2BY, CB2BCD, CFP2I, CI2FP, CNVBFP.

Name

Convert a binary-coded decimal value to ASCII characters

Description

Although the processor has a decimal flag and can perform math in binary-coded decimal (BCD), this mode is rarely used on Commodore computers. The CIA chips' time-of-day clocks keep time in BCD format, but that's about it.

If you decide there's some merit in using BCD math, however, this routine will convert a single BCD byte into two ASCII numbers. Its construction closely resembles the conversion routine that handles hexadecimal.

Prototype

1. Enter with the number to be converted in the accumulator.
2. Save it temporarily.
3. AND with the number $0F and add 48 for the low nybble.
4. Transfer to .X.
5. Restore the previous value.
6. Repeat the above steps, but rotate right four times with the carry flag set (or cleared) as needed to add 48.
7. Exit with the high nybble in .A, the low in .X.

Explanation

The high and low nybbles of a byte are the top four bits and the bottom four, respectively. A nybble is half a byte. Normally, a nybble can have 16 possible settings, from %0000 through %1111. Given two nybbles, a byte can hold 256 possible values (16 × 16). Not so in decimal mode. If you set the decimal flag (with the SED operation), nybbles are suddenly limited to 10 values, from $0 through $9. That means bytes can hold only 100 different numbers (10 × 10), from $00 through $99.

Such mathematical operations as addition (ADC) and subtraction (SBC) are also affected by the decimal flag. Suddenly, $35 plus $49 is $84 (in decimal mode) instead of $7E (in nondecimal mode). The number $2001 in hex means 8193. But in decimal mode, $2001 means, well, 2001. For those of us who count with ten fingers, decimal mode is quite convenient.

If you need to print out a BCD number, this routine will do the trick. It basically isolates the nybbles and adds 48 to convert one byte into two ASCII characters, which can then be printed.

Routine

```
C000                CHROUT  =    $FFD2
C000  A9 93                 LDA  #$93          ; convert $93
C002  20 0D C0              JSR  BCD2AX        ; to the characters 9 and 3
C005  20 D2 FF              JSR  CHROUT        ; print 9
C008  8A                    TXA
C009  20 D2 FF              JSR  CHROUT        ; print 3
C00C  60                    RTS
                                               ;
C00D  D8            BCD2AX  CLD                ; make sure decimal mode is off
C00E  48                    PHA                ; save the value
C00F  29 0F                 AND  #%00001111    ; low nybble first
C011  09 30                 ORA  #48           ; add 48 for ASCII
C013  AA                    TAX                ; result in .X (or you can store it in
                                               ; memory)
C014  68                    PLA                ; get back the original value
C015  29 F0                 AND  #%11110000    ; high nybble
C017  38                    SEC                ; what will become bit 5 (16)?
C018  6A                    ROR                ; move it right one
C019  38                    SEC                ; bit 6 (32)
C01A  6A                    ROR                ; right again
C01B  4A                    LSR
C01C  4A                    LSR                ; and shift right with zeros
C01D  60                    RTS                ; done (high nybble in .A, low in .X)
```

See also CAS2IN, CB2ASC, CB2HEX, CI2HEX.

Name

Convert binary-coded de imal (BCD) to a byte value

Description

If you need to convert a binary-coded decimal (BCD) number to a standard byte value, this routine will do it.

Prototype

1. Store the value temporarily in memory.
2. Get the high nybble by masking off the low nybble.
3. Shift the high nybble right once (the nybble value times eight). Store it in the RESULT byte.
4. Shift it right twice more (nybble times two). Add the number to RESULT.
5. Reload the original value.
6. Mask off the high nybble.
7. Add the low nybble to RESULT.

Explanation

The SED (SEt Decimal) operation puts the 64 and 128 into decimal mode, where the accumulator can hold only 100 values instead of 256. Each nybble counts from $0 through $9 instead of $0 through $F. Thus, if you add $03 to $19, the result is $22 instead of $1C (because 3 plus 19 is 22 in decimal arithmetic).

Converting a BCD number to a normal byte value means changing a number like $71 to $47, because 71 in decimal is $47 in hexadecimal. The ten's place of $71 is the high nybble, $7. If the low byte is masked off, the number becomes $70 (decimal 112). Shift it right once and it becomes $38 (decimal 56), which is 8 × 7. That number gets stored in memory. Shift it right two more times, and $38 is changed to $0E (decimal 14), which is two 2 × 7. Add that to the first number, and the result is decimal 70, because (8 × 7) + (2 × 7) is the same as 10 × 7. This operation changes $70 (112) to 70 ($46). The next step is to add in the low nybble, the one's place in both decimal and hexadecimal.

Routine

```
C000                CHROUT   =     $FFD2
C000                LINPRT   =     $BDCD     ; LINPRT = $8E32 on the 128
                                             ;
C000  A0 00         FRAME    LDY   #0
C002  8C 1F  C0     LOOP     STY   TEMPY
C005  B9 20  C0              LDA   LIST,Y    ; get a BCD value
C008  20 25  C0              JSR   BCD2BY    ; convert it
C00B  AA                     TAX
```

```
C00C  A9 00              LDA   #0
C00E  20 CD BD           JSR   LINPRT         ; print it
C011  A9 0D              LDA   #13
C013  20 D2 FF           JSR   CHROUT         ; and a RETURN
C016  AC 1F C0           LDY   TEMPY          ; get .Y back
C019  C8                 INY                  ; INC it
C01A  C0 05              CPY   #5             ; is it 5 yet?
C01C  D0 E4              BNE   LOOP           ; no, go back
C01E  60                 RTS                  ; else, end
                                              ;
C01F  00         TEMPY   .BYTE 0
C020  10 01 99   LIST    .BYTE $10,$01,$99,$50,$55
                                              ;
C025  D8         BCD2BY  CLD                  ; just to be sure that decimal mode isn't on
C026  8D 45 C0           STA   TEMPA          ; save the number
C029  29 F0              AND   #%11110000     ; get the high nybble
C02B  4A                 [redacted]           ; shift right (nybble × 16)/2 is nybble × 8
C02C  8D 46 C0           STA   RESULT         ; start preparing the result
C02F  4A                 LSR                  ; nybble × 4
C030  4A                 LSR                  ; nybble × 2
C031  18                 CLC                  ; now add nybble × 8 and nybble × 2
C032  6D 46 C0           ADC   RESULT         ; which is nybble × (8 + 2)
C035  8D 46 C0           STA   RESULT         ; and we're almost done
C038  AD 45 C0           LDA   TEMPA          ; now the low nybble
C03B  29 0F              AND   #%00001111     ; get the four bits
C03D  18                 CLC
C03E  6D 46 C0           ADC   RESULT         ; add it
C041  8D 46 C0           STA   RESULT         ; store it, for whatever reason
C044  60                 RTS                  ; all done
                                              ;
C045  00         TEMPA   .BYTE 0
C046  00         RESULT  .BYTE 0
```

See also B2SNIN, B2UNIN, CB2BCD, CFP2I, CI2FP, CNVBFP.

Name

Set the text screen background color

Description

This routine sets the background color of the text screen. Pick a color value, assign it as COLVAL, and access the routine.

Prototype

1. Enter this routine with the selected background color in .A.
2. Store .A in the background color register at 53281 (BGCOL0).

Explanation

The example program shows how to set the background color of the screen to red. Here, COLVAL is given a value of 2, representing the color red. To choose another color, use the table of color values found under **COLFIL**.

Routine

```
C000             BGCOL0  =    53281     ; background color register 0
                                        ;
                                        ; Set background to red.
C000  AD 0B C0           LDA  COLVAL    ; .A contains screen background color
C003  20 07 C0           JSR  BCKCOL    ; set it
C006  60                 RTS
                                        ;
                                        ; Set background color. Color value in .A
C007  8D 21 D0  BCKCOL   STA  BGCOL0    ; set background
C00A  60                 RTS
                                        ;
C00B  02        COLVAL   .BYTE 2        ; color red
```

See also BORCOL, COLFIL, TXTCCH, TXTCOL.

Name

Emit a beep sound

Description

BEEPER produces a beep. Call it whenever you want to get the user's attention without startling him or her. You could use it, for example, to prompt for a question or to signal a correct (or incorrect) response.

Prototype

1. Clear the SID chip with **SIDCLR**.
2. Set up the necessary SID chip parameters for voice 1. Set volume to 15, attack/decay to 0, sustain/release to $F0, low frequency to 132, and high frequency to 125.
3. Select a triangle waveform for voice 1 and start the attack/decay/sustain cycle (set the gate bit).
4. Allow a delay of two jiffies and then start the release cycle (clear the gate bit).

Explanation

Depending upon the application, the beeping sound that **BEEPER** generates may or may not be quite what you're looking for. If it's not what you want, experiment with the SID chip parameters in the routine until you get the effect you want.

When the SID chip is called upon to make a particular sound, it often echoes the last frequency at a level that is barely audible even after the release cycle is complete. In fact, this occurs to some degree with **BEEPER**. If you find this effect annoying, you can stop it before exiting from the routine. Either store zeros in the frequency registers (FRELO1, FREHI1), or simply turn the chip off altogether by JSRing to **SIDCLR**.

Routine

```
C000                    SIGVOL   =   54296      ; SID chip volume register
C000                    ATDCY1   =   54277      ; voice 1 attack/decay register
C000                    SUREL1   =   54278      ; voice 1 sustain/release register
C000                    FRELO1   =   54272      ; voice 1 frequency control (low byte)
C000                    FREHI1   =   54273      ; voice 1 frequency control (high byte)
C000                    VCREG1   =   54276      ; voice 1 control register
C000                    JIFFLO   =   162        ; low byte of jiffy clock
                                                ;
C000  20 2F C0  BEEPER  JSR      SIDCLR         ; clear the SID chip
C003  A9 0F             LDA      #15            ; set the volume
C005  8D 18 D4          STA      SIGVOL
C008  A9 00             LDA      #$0            ; set attack/decay
C00A  8D 05 D4          STA      ATDCY1
C00D  A9 F0             LDA      #$F0           ; set sustain/release
```

```
C00F  8D 06 D4          STA  SUREL1
C012  A9 84             LDA  #132          ; set voice 1 frequency (low byte)
C014  8D 00 D4          STA  FRELO1
C017  A9 7D             LDA  #125          ; set voice 1 frequency (high byte)
C019  8D 01 D4          STA  FREHI1
C01C  A9 11             LDA  #%00010001    ; select triangle waveform and gate sound
C01E  8D 04 D4          STA  VCREG1
C021  A9 02             LDA  #2            ; cause a delay of two jiffies
C023  65 A2             ADC  JIFFLO        ; add current jiffy reading
C025  C5 A2     DELAY   CMP  JIFFLO        ; and wait for two jiffies to elapse
C027  D0 FC             BNE  DELAY
C029  A9 10             LDA  #%00010000    ; ungate sound
C02B  8D 04 D4          STA  VCREG1
C02E  60                RTS
                                           ;
                                           ; Clear the SID chip.
C02F  A9 00     SIDCLR  LDA  #0            ; fill with zeros
C031  A0 18             LDY  #24           , index to FRELO1
C033  99 00 D4  SIDLOP  STA  FRELO1,Y      ; store zero in SID chip address
C036  88                DEY                ; for next lower byte
C037  10 FA             BPL  SIDLOP        ; fill 25 bytes
C039  60                RTS
```

See also BELLRG, EXPLOD, INTMUS, MELODY, NOTETB, SIDCLR, SIDVOL, SIRENS.

Name

Emit a bell sound

Description

BELLRG produces a bell tone. You might find it useful in your programs as a signal to the user that some ongoing task—like copying a memory buffer to disk—has finished.

Prototype

1. Clear the SID chip with **SIDCLR**.
2. Set up the necessary SID chip parameters. Set volume to 7, attack/decay and sustain/release of voice 1 to $0A, and the high frequency of both voices 1 and 3 to 67.
3. Select a triangle waveform for voice 1. At the same time, set bit 2 for ring modulation and start the attack/decay/sustain cycle (set the gate bit).
4. Start the release cycle of voice 1 (clear the gate bit).

Explanation

This routine relies on *ring modulation* to simulate a bell sound. Ring modulation produces a waveform that is a combination of the sum and difference of two waveforms of different frequencies.

You can use any or all of the SID chip's three voices for ring modulation. In **BELLRG**, the frequency of voice 1 is ring modulated by the selection of a triangle waveform for this voice and by storage of a second frequency value in voice 3. Since voice 3 is not actually heard, no SID chip parameters other than the frequency value are necessary for this voice. Here, identical frequencies are used for both voices.

Storing different frequencies in voice 3 will produce widely varying sound effects. For instance, a 10 in FREHI3 will cause a gonglike sound rather than a bell. To set this up, insert an LDA #10 instruction just before the STA FREHI3 at $C015.

The SID chip often tends to run on in the background even after the release cycle is complete. **BELLRG** is not immune from this effect. To stop this from happening, store zeros in the frequency registers (FREHI1, FREHI3), or turn off the chip altogether by JSRing to **SIDCLR** once the bell has sounded.

BELLRG

Routine

```
C000                SIGVOL    =     54296         ; SID chip volume register
C000                ATDCY1    =     54277         ; voice 1 attack/decay register
C000                SUREL1    =     54278         ; voice 1 sustain/release register
C000                FRELO1    =     54272         ; voice 1 frequency control (low byte)
C000                FREHI1    =     54273         ; voice 1 frequency control (high byte)
C000                FREHI3    =     54287         ; voice 3 frequency (high byte)
C000                VCREG1    =     54276         ; voice 1 control register
                                                  ;
C000  20 23 C0      BELLRG    JSR   SIDCLR        ; clear the SID chip
C003  A9 07                   LDA   #7            ; set the volume
C005  8D 18 D4                STA   SIGVOL
C008  A9 0A                   LDA   #$0A          ; set attack/decay
C00A  8D 05 D4                STA   ATDCY1
C00D  8D 06 D4                STA   SUREL1        ; set sustain/release
C010  A9 43                   LDA   #67           ; set voice 1 high frequency
C012  8D 01 D4                STA   FREHI1
C015  8D 0F D4                STA   FREHI3        ; for ring modulation
C018  A9 15                   LDA   #%00010101    ; select triangle waveform/ring
                                                  ; modulation/gate the sound
C01A  8D 04 D4                STA   VCREG1
C01D  A9 14                   LDA   #%00010100    ; ungate the sound
C01F  8D 04 D4                STA   VCREG1
C022  [redacted]              RTS
                                                  ;
                                                  ; Clear the SID chip.
C023  A9 00         SIDCLR    LDA   #0            ; fill with zeros
C025  A0 18                   LDY   #24           ; index to FRELO1
C027  99 00 D4      SIDLOP    STA   FRELO1,Y      ; store zero in each SID chip address
C02A  88                      DEY                 ; for next lower byte
C02B  10 FA                   BPL   SIDLOP        ; fill 25 bytes
C02D  60                      RTS
```

See also BEEPER, EXPLOD, INTMUS, MELODY, NOTETB, SIDCLR, SIDVOL, SIRENS.

Name

Display in a virtual window portions of a much larger map

Description

The normal 40-column screen, with 25 rows, is somewhat limited when it comes to games or applications that need a larger workspace. This routine allows you to use the 40-column screen as a window on a larger screen.

Prototype

1. Set aside a section of memory for use as the big screen.
2. Place values for the upper left corner in CORNRX and CORNRY.
3. Establish a zero-page pointer for the real screen.
4. Working 40 characters at a time, store a character and a color into screen and color memory.
5. At the end of each line, add 40 to the zero-page pointer.
6. Add the width of the large screen to that pointer.
7. While the number of rows is less than maximum, continue to loop back to step 4.

Explanation

Although this routine is great for war games and adventure games (both of which benefit when they have a large map area), it could also be used in a serious application like a spreadsheet.

The example map is 100 columns by 50 rows. You can adjust this by changing the variables WIDTH and HEIGHT at $C120-$C121. The variables LINES and COLS indicate the size of the normal text screen.

Note that 100 columns and 50 rows give you 5000 cells on the large map. This means the program uses 5000 bytes of memory. The larger you make the map, the more memory it needs. If you create your own map, you could load it into memory directly from disk. The example uses a table to build the map. The label CRUNCH5 at $C149 contains four numbers: 80, 1, 20, and 2. This means line 5 of the large screen contains 80 ones and 20 twos. There are only five characters allowed on this particular map (a maximum of 256 can be placed, if you expand the table MCHAR and MCOLR just before the CRUNCH table).

The five characters are:

0	White	102	crosshatch
1	Green	88	spade
2	Blue	160	RVS space
3	Black	81	ball
4	Gray2	87	circle

The numbers in MCHAR ($C129) are screen codes. In MCOLR ($C12E), the numbers are color codes. In the 5000 bytes of the map, you'll find the numbers 0–4. When a portion of the map is displayed, the number is used as an index into MCHAR and MCOLR, and the corresponding numbers are POKEd to screen or color memory.

The framing routine looks for the cursor keys (up, down, left, and right) and moves the values of CORNRX and CORNRY according to the direction of movement. You won't have to scroll one character at a time, however. Just store new values to CORNRX and CORNRY and call **BIGMAP**. To exit this routine, press RETURN.

Note: Since location $4000 is in bank 0 on the 128, you may want to put the map at $2000 instead. If you use a 128, you should substitute MAPTAB = $2000 in the list of equates at the beginning of the program.

Routine

```
C000                ZP        =     $F9
C000                ZS        =     $FB
C000                ZC        =     $FD
C000                GETIN     =     $FFE4
C000                SCREEN    =     $0400      ; screen memory
C000                COLOR     =     $D800      ; color memory
C000                MAPTAB    =     $4000      ; lookup table for map
                                               ;
C000  20 DF C0                JSR   MAKMAP     ; uncrunch the map
C003  20 62 C0                JSR   BIGMAP     ; print the map (starting at 0,0)
C006  20 E4 FF  GLP           JSR   GETIN      ; get a key
C009  F0 FB                   BEQ   GLP
C00B  C9 0D                   CMP   #13        ; is it RETURN?
C00D  D0 01                   BNE   MORE       ;
C00F  60                      RTS              ; yes, so quit
                                               ;
C010  C9 11     MORE          CMP   #17        ; if cursor down
C012  F0 1B                   BEQ   MOVEDN     ; move the map down
C014  C9 91                   CMP   #145       ; if cursor up
C016  F0 29                   BEQ   MOVEUP     ; move the map up
C018  C9 1D                   CMP   #29        ; cursor right
C01A  F0 34                   BEQ   MOVERT
C01C  C9 9D                   CMP   #157       ; check cursor left
C01E  D0 E6                   BNE   GLP        ; if not left, go back
C020  AE 24 C1  MOVELF        LDX   CORNRX     ; get the x corner
C023  F0 E1                   BEQ   GLP        ; if zero, it can't decrement
C025  CA                      DEX              ; else, count down
```

```
C026  8E 24 C1          STX   CORNRX
C029  20 62 C0          JSR   BIGMAP
C02C  4C 06 C0          JMP   GLP
C02F  AC 25 C1  MOVEDN  LDY   CORNRY      ; change the y corner
C032  CC 27 C1          CPY   MAXY        ; is it at the top value?
C035  F0 CF             BEQ   GLP         ; yes, skip it
C037  C8                INY               ; else, add one
C038  8C 25 C1          STY   CORNRY
C03B  20 62 C0          JSR   BIGMAP
C03E  4C 06 C0          JMP   GLP
                                          ;
C041  AC 25 C1  MOVEUP  LDY   CORNRY      ; check the y location
C044  F0 C0             BEQ   GLP         ; if zero, skip it
C046  88                DEY               ; count back one
C047  8C 25 C1          STY   CORNRY
C04A  20 62 C0          JSR   BIGMAP
C04D  4C 06 C0          JMP   GLP
                                          ;
C050  AE 24 C1  MOVERT  LDX   CORNRX      ; increment the x corner
C053  EC 26 C1          CPX   MAXX        , is it the maximum?
C056  F0 AE             BEQ   GLP         ; if so, go back
C058  E8                INX
C059  8E 24 C1          STX   CORNRX
C05C  20 62 C0          JSR   BIGMAP
C05F  4C 06 C0          JMP   GLP
                                          ;
C062  A9 00     BIGMAP  LDA   #<MAPTAB    ; set up ZP to point to the map table
C064  85 F9             STA   ZP
C066  A9 40             LDA   #>MAPTAB
C068  85 FA             STA   ZP+1
C06A  AC 25 C1  FIXROW  LDY   CORNRY      ; row number
C06D  F0 0F             BEQ   FIXCOL      ; if row 0, skip ahead
C06F  18        LPROW   CLC               ; else, add the number of columns
C070  A5 F9             LDA   ZP          ; to the pointer
C072  6D 20 C1          ADC   WIDTH
C075  85 F9             STA   ZP
C077  90 02             BCC   INROW       ; if the carry flag is set
C079  E6 FA             INC   ZP+1        ; then increment the high byte
C07B  88        INROW   DEY               ; count down
C07C  D0 F1             BNE   LPROW       ; and loop back
                                          ;
C07E  AD 24 C1  FIXCOL  LDA   CORNRX      ; now add the x offset
C081  18                CLC
C082  65 F9             ADC   ZP          ; add to ZP
C084  85 F9             STA   ZP          ; store it
C086  A9 00             LDA   #0          ; fix the high byte
C088  65 FA             ADC   ZP+1        ; add zero or one
C08A  85 FA             STA   ZP+1        ; depending on whether carry is set or not
                                          ;
                                          ; Now the pointer ZP is set up.
                                          ; Set up a second pointer to the screen and
                                          ; color memory.
C08C  A9 00             LDA   #<SCREEN
C08E  85 FB             STA   ZS
C090  A9 04             LDA   #>SCREEN
C092  85 FC             STA   ZS+1
C094  A9 00             LDA   #<COLOR
C096  85 FD             STA   ZC
C098  A9 D8             LDA   #>COLOR
C09A  85 FE             STA   ZC+1
                                          ;
                                          ; Start storing the characters and colors.
C09C  AD 22 C1          LDA   LINES       ; number of lines
C09F  8D 28 C1          STA   COUNTR      ; COUNTR will count down
```

```
C0A2  AC 23 C1  STORLP  LDY   COLS         ; number of columns
C0A5  B1 F9     INLOOP  LDA   (ZP),Y       ; get the character number
C0A7  AA                TAX                ; which is an offset
C0A8  BD 29 C1          LDA   MCHAR,X      ; to the character
C0AB  91 FB             STA   (ZS),Y       ; store it to the screen
C0AD  BD 2E C1          LDA   MCOLR,X      ; also, a color
C0B0  91 FD             STA   (ZC),Y       ; which goes in color memory
C0B2  88                DEY                ; .Y counts down
C0B3  10 F0             BPL   INLOOP       ; 40 times (in this example)
                                           ; After each time through the loop,
                                           ; fix the zero-page pointers.
C0B5  18                CLC
C0B6  A5 F9             LDA   ZP           ; to ZP
C0B8  6D 20 C1          ADC   WIDTH        ; add the width of the big map
C0BB  85 F9             STA   ZP
C0BD  A9 00             LDA   #0
C0BF  65 FA             ADC   ZP+1         ; add zero or one
C0C1  85 FA             STA   ZP+1         ; to ZP+1
C0C3  18                CLC
C0C4  A5 FB             LDA   ZS           ; to ZS
C0C6  69 28             ADC   #40          ; add 40
C0C8  85 FB             STA   ZS           ; and store it
C0CA  90 02             BCC   FC
C0CC  E6 FC             INC   ZS+1
C0CE  18        FC      CLC
C0CF  A5 FD             LDA   ZC           ; to ZC
C0D1  69 28             ADC   #40          ; add 40
C0D3  85 FD             STA   ZC
C0D5  90 02             BCC   FD
C0D7  E6 FE             INC   ZC+1
C0D9  CE 28 C1  FD      DEC   COUNTR       ; now see if it's time to leave
C0DC  10 C4             BPL   STORLP       ; no, do another row
C0DE  60                RTS
                                           ;
C0DF  A9 00     MAKMAP  LDA   #<MAPTAB     ; set up ZP to point to the table
C0E1  85 F9             STA   ZP
C0E3  A9 40             LDA   #>MAPTAB
C0E5  85 FA             STA   ZP+1
C0E7  A9 33             LDA   #<CRUNCH0    ; and ZS points to the crunch table
C0E9  85 FB             STA   ZS
C0EB  A9 C1             LDA   #>CRUNCH0
C0ED  85 FC             STA   ZS+1
                                           ;
C0EF  A0 00     MAKLP   LDY   #0
C0F1  B1 FB             LDA   (ZS),Y       ; number of times to loop
C0F3  F0 2A             BEQ   MKQUIT       ; quit if zero
C0F5  AA                TAX                ; put it in .X
C0F6  8D 28 C1          STA   COUNTR       ; save in COUNTR, too
C0F9  C8                INY
C0FA  B1 FB             LDA   (ZS),Y       ; the fill character is in .A
C0FC  88                DEY                ; .Y is back to zero
C0FD  91 F9     MKSTOR  STA   (ZP),Y       ; store it in MAPTAB memory
C0FF  C8                INY                ; .Y counts forward
C100  CA                DEX                ; .X counts down
C101  D0 FA             BNE   MKSTOR       ; loop
                                           ;
                                           ; Now fix ZP and ZS.
C103  A5 FB             LDA   ZS
C105  18                CLC
C106  69 02             ADC   #2           ; add 2
C108  85 FB             STA   ZS
C10A  90 02             BCC   AHD
C10C  E6 FC             INC   ZS+1
C10E  A5 F9     AHD     LDA   ZP
```

```
C110  18              CLC
C111  6D  28  C1      ADC    COUNTR
C114  85  F9          STA    ZP
C116  A9  00          LDA    #0
C118  65  FA          ADC    ZP+1
C11A  85  FA          STA    ZP+1
C11C  4C  EF  C0      JMP    MAKLP
C11F  60      MKQUIT  RTS
                                              ;
C120  64      WIDTH    .BYTE  100             ; width of the big map
C121  32      HEIGHT   .BYTE  50              ; height of the big screen
C122  18      LINES    .BYTE  24              ; number of screen lines (0-24 is 25 lines)
C123  27      COLS     .BYTE  39              ; number of screen columns (0-39 is total
                                              ; of 40)
C124  00      CORNRX   .BYTE  0               ; x-position of upper left corner
C125  00      CORNRY   .BYTE  0               ; y-position of corner
C126  3C      MAXX     .BYTE  60
C127  19      MAXY     .BYTE  25
C128  00      COUNTR   .BYTE  0
                                              ;
C129  66  58  A0  MCHAR     .BYTE  102,88,160,81,87
C12E  01  05  06  MCOLR     .BYTE  1,5,6,0,12
C133  64  00      CRUNCH0   .BYTE  100,0
C135  31  00  01  CRUNCH1   .BYTE  49,0,1,1,50,0
C13B  0A  00  50  CRUNCH2   .BYTE  10,0,80,1,10,0
C141  64  01      CRUNCH3   .BYTE  100,1
C143  0E  01  02  CRUNCH4   .BYTE  14,1,2,3,84,1
C149  50  01  14  CRUNCH5   .BYTE  80,1,20,2
C14D  52  01  12  CRUNCH6   .BYTE  82,1,18,2
C151  53  01  01  CRUNCH7   .BYTE  83,1,1,4,16,2
C157  0F  00  45  CRUNCH8   .BYTE  15,0,69,1,16,2
C15D  1E  00  37  CRUNCH9   .BYTE  30,0,55,1,15,2
C163  32  00  26  CRUNCH10  .BYTE  50,0,38,1,12,2
C169  34  00  24  CRUNCH11  .BYTE  52,0,36,1,12,2
C16F  58  00  01  CRUNCH12  .BYTE  88,0,1,3,11,2
C175  5A  00  0A  CRUNCH13  .BYTE  90,0,10,2
C179  57  00  0D  CRUNCH14  .BYTE  87,0,13,2
C17D  52  00  12  CRUNCH15  .BYTE  82,0,18,2
C181  05  00  01  CRUNCH16  .BYTE  5,0,1,4,5,0,1,4,63,0,25,2
C18D  02  01  49  CRUNCH17  .BYTE  2,1,73,0,25,2
C193  06  01  4A  CRUNCH18  .BYTE  6,1,74,0,20,2
C199  0A  01  4B  CRUNCH19  .BYTE  10,1,75,0,15,2
C19F  0E  01  47  CRUNCH20  .BYTE  14,1,71,0,15,2
C1A5  12  01  01  CRUNCH21  .BYTE  18,1,1,4,67,0,14,2
C1AD  17  01  14  CRUNCH22  .BYTE  23,1,20,0,1,3,47,0,9,2
C1B7  1F  01  45  CRUNCH23  .BYTE  31,1,69,0
C1BB  23  01  41  CRUNCH24  .BYTE  35,1,65,0
C1BF  24  01  01  CRUNCH25  .BYTE  36,1,1,4,63,0
C1C5  20  01  44  CRUNCH26  .BYTE  32,1,68,0
C1C9  18  01  4C  CRUNCH27  .BYTE  24,1,76,0
C1CD  0A  01  5A  CRUNCH28  .BYTE  10,1,90,0
C1D1  64  00      CRUNCH29  .BYTE  100,0
C1D3  64  00      CRUNCH30  .BYTE  100,0
C1D5  50  00  14  CRUNCH31  .BYTE  80,0,20,1
C1D9  45  00  02  CRUNCH32  .BYTE  69,0,2,3,29,1
C1DF  33  00  01  CRUNCH33  .BYTE  51,0,1,3,48,1
C1E5  33  00  31  CRUNCH34  .BYTE  51,0,49,1
C1E9  2D  00  37  CRUNCH35  .BYTE  45,0,55,1
C1ED  27  00  05  CRUNCH36  .BYTE  39,0,5,1,1,4,55,1
C1F5  20  00  44  CRUNCH37  .BYTE  32,0,68,1
C1F9  14  00  50  CRUNCH38  .BYTE  20,0,80,1
C1FD  12  00  1E  CRUNCH39  .BYTE  18,0,30,1,1,3,51,1
C205  0F  00  55  CRUNCH40  .BYTE  15,0,85,1
```

```
C209  0D  00  57  CRUNCH41 .BYTE  13,0,87,1
C20D  0B  00  59  CRUNCH42 .BYTE  11,0,89,1
C211  64  01      CRUNCH43 .BYTE  100,1
C213  5A  01  0A  CRUNCH44 .BYTE  90,1,10,2
C217  50  01  14  CRUNCH45 .BYTE  80,1,20,2
C21B  3C  01  28  CRUNCH46 .BYTE  60,1,40,2
C21F  32  01  01  CRUNCH47 .BYTE  50,1,1,3,49,2
C225  29  01  3B  CRUNCH48 .BYTE  41,1,59,2
C229  14  01  50  CRUNCH49 .BYTE  20,1,80,2
C22D  00  00               .BYTE  0,0            ; end of the table is a zero
```

See also WINDOW.

Name

Enable/disable the hi-res screen (bitmap mode)

Description

This routine turns on the hi-res screen if it's off and turns it off if it's currently on.

Prototype

EOR the contents of SCROLY (or GRAPHM on the 128) with %00100000 and store the result back in the appropriate register.

Explanation

On the 64, setting bit 5 of the vertical fine-scrolling/control register at 53265 (labeled SCROLY) enables high-resolution graphics, or bitmap mode. On the 128, GRAPHM (location 216) serves as a shadow register for SCROLY. During each IRQ interrupt of the 128, the contents of GRAPHM are copied to SCROLY. So, to enable bitmap mode on the 128, set bit 5 of location 216.

To disable bitmap mode and return to the normal text-screen arrangement, clear bit 5 of either SCROLY on the 64 or GRAPHM on the 128.

Both operations, enabling and disabling bitmap mode, can be carried out by exclusive-ORing this bit.

Routine

```
C000              SCROLY  =     53265       ; scroll/control register; use GRAPHM = 216
                                            ; on the 128
                                            ;
                                            ; Enable/disable bitmap mode
C000  AD 11  D0  BITMAP  LDA   SCROLY      ; substitute GRAPHM for SCROLY on the 128
C003  49 20              EOR   #%00100000  ; flip bit 5
C005  8D 11  D0          STA   SCROLY      ; reset register (again use GRAPHM instead of
                                            ; SCROLY on the 128
C008  60                 RTS
```

See also SCRDN1, SCRDN2, SCRDN3; CLRHRF or CLRHRS for example programs using BITMAP.

Name

Set the text screen border color

Description

BORCOL uses the color value in the accumulator to set the border color of the text screen. A table of color values and their corresponding colors is given under **COLFIL**.

Prototype

1. Come into this routine with the designated border color value in .A.
2. Store .A in the border color register at 53280 (EXTCOL).

Explanation

In the example program, the border color of the screen continually cycles through the 16 available colors. Pressing any key exits the routine.

A series of horizontal, or *raster*, lines make up the screen display. These raster lines are updated and redrawn every 1/60 second. Only 200 (lines 50–249) of the 262 raster lines (312 on European machines) are actually part of the visible display. The rest constitute the screen border.

Here, we determine the current raster line being drawn with RASTER, changing the border color only when this raster line is off the top of the visible screen area (when it has a value of 25 or less). This prevents the "moving lines" effect where the raster line is updated before it's completely drawn.

Routine

```
C000              EXTCOL  =    53280     ; border color register
C000              RASTER  =    53266     ; current raster scan line
C000              GETIN   =    65508
C000              ZP      =    251
                                         ;
                                         ; Cycle border color while raster line is off
                                         ; bottom of screen.
C000  AD 12 D0 GETRAS     LDA  RASTER    ; check current raster line
C003  C9 19               CMP  #25       ; is it off the top of the screen?
C005  90 F9               BCS  GETRAS    ; no, so wait
C007  E6 FB               INC  ZP        ; yes, so cycle color
C009  A5 FB               LDA  ZP        ; .A contains border color
C00B  20 14 C0            JSR  BORCOL    ; change it
C00E  20 E4 FF WAIT       JSR  GETIN     ; get a keypress
C011  F0 ED               BEQ  GETRAS    ; no key, so continue to cycle
C013  60                  RTS
                                         ;
                                         ; Set border color. .A holds color value.
C014  8D 20 D0 BORCOL     STA  EXTCOL    ; set register
C017  60                  RTS
```

See also BCKCOL, COLFIL, TXTCCH, TXTCOL.

Name

Clear the keyboard buffer

Description

There are often situations where you want to accept only the last input from the user and ignore any previous input. For instance, suppose your program has a series of yes/no questions requiring a Y or N response. If the user's finger lingers on a key, several such responses can unintentionally be entered into the keyboard buffer. And subsequent questions will be answered, for better or for worse, in a flash.

Or suppose you have written a game that requires keyboard control. At the end of the game, you might have a "Play again (Y/N)?" question. If the keyboard buffer contains a number of moves, the question can be answered before the player realizes what has happened.

In both cases, you need to clear the keyboard buffer just before a response is accepted. To clear the keyboard buffer, simply store a zero in NDX, the location containing the number of characters currently in the buffer.

Prototype

Store a zero in the keyboard buffer character counter, NDX.

Explanation

The example routine illustrates how to clear the keyboard buffer before input is accepted.

The keyboard buffer, which begins at location 631 on the 64 and location 842 on the 128, can hold up to ten characters before overflow occurs. When the buffer fills, additional characters are ignored. Note that GETIN returns the first character placed in the buffer.

Routine

```
C000                NDX     =      198      ; NDX = 208 on the 128—number of characters
                                            ; in keyboard buffer
C000                GETIN   =      65508
C000                CHROUT  =      65490
                                            ;
                                            ; Clear keyboard buffer and fetch a keypress.
C000  20 0C C0              JSR    BUFCLR   ; clear the keyboard buffer
C003  20 E4 FF      WAIT    JSR    GETIN    ; fetch the next character
C006  F0 FB                 BEQ    WAIT     ; no keypress, so WAIT
C008  20 D2 FF              JSR    CHROUT   ; print it
C00B  60                    RTS
                                            ;
                                            ; Clear the keyboard buffer.
C00C  A9 00         BUFCLR  LDA    #0
C00E  85 C6                 STA    NDX      ; set number of keys to 0
C010  60                    RTS
```

See also CHRGTR, CHRGTS, CHRKER, MATGET.

Name

Cause a one-byte delay

Description

Access the **BYT1DL** routine whenever you need to produce a very brief delay in your program. By using this routine, you can generate delays of a millisecond or less.

Prototype

1. Enter this routine with the delay byte in .X.
2. Decrement .X to zero and then RTS.

Explanation

The requirements of the routine are simple: Just load the X register with a delay value—some number from 0–255—and JSR to the routine.

Within the routine itself, a branching loop repeats until .X decrements to zero. Because .X is decremented before the branch, the maximum delay occurs when .X is initially equal to zero. In this case, 256 branches take place.

By knowing the number of machine cycles required by each instruction (see the opcode table which appears elsewhere in this book), you can determine the actual delay time for **BYT1DL** based on the incoming .X value. Within the routine, each DEX requires two cycles while the BNE, assuming no page boundaries are crossed, takes three. If no branch actually occurs, which is the case on the last pass through the loop, the BNE instruction requires only two cycles.

In addition to instructions within the loop, you must consider the JSR and the RTS. Both of these require six cycles.

Overall, then, the number of machine cycles (MC), based on the incoming .X value, can be calculated by using the formula $MC = B * X - 1 + 12$. B is either 5 or 6 here, depending on whether or not the branch crosses a page boundary; X is the number of times the loop repeats. In all cases but one—the exception being when .X is initially zero—X in the formula is the same as the contents of the X register.

On the 64 or 128, the duration of each machine cycle is based on the clock speed for the microprocessor. For North American (NTSC) systems, the microprocessor runs at 1,022,730 Hz (cycles per second). European systems (PAL) have a clock speed of 985,250 Hz. At either rate, each machine cycle takes approximately 1 microsecond (1E−6 sec-

ond)—0.978 microseconds for NTSC systems and 1.015 microseconds for PAL systems.

And so, if .X were zero coming into **BYT1DL**, the delay loop would have a maximum number of repetitions—256. In this case, a delay of 5 * 256 − 1 + 12, or 1291, machine cycles would result. Assuming a 64 or 128 using the North American convention, the actual time that elapses would be 1291*0.978 microseconds, or 1.263 milliseconds (1.263E−3 second).

On the other hand, if .X holds 1 upon entry into the loop—so no repetitions take place—a delay of 16 cycles, or 15.6 microseconds, would result.

All in all, then, **BYT1DL** offers a wide range of delays, although they're consistently brief. If you need to, you can adjust the range of these delays upward by inserting additional instructions between the DEX and the BNE instruction in the loop. Just make sure any instructions you add don't affect the execution of the routine.

A typical practice is to insert one or more NOPs, which take two cycles each, in the code. Of course, you could also use instructions other than the NOP here, as long as they have no effect on the zero flag. For instance, inserting an STX, which stores into an unused absolute address, would add four cycles each time through the loop.

Routine

```
C000              BGCOL0   =    53281    ; screen background color
C000              DELAY    =    255      ; one-byte delay value
                                         ;
                                         ; Set the screen background color to light
                                         ; gray, cause a one-byte delay
                                         ; (based on .X), and then change the
                                         ; background color to black.
C000  A9 0F       MAIN     LDA  #15      ; for light gray background
C002  8D 21 D0    BCKCOL   STA  BGCOL0
C005  A2 FF                LDX  #DELAY   ; enter BYT1DL with the delay value in .X
C007  20 0E C0             JSR  BYT1DL   ; cause a delay
C00A  EE 21 D0             INC  BGCOL0   ; to produce a black background (only low
                                         ; nybble is significant)
C00D  60                   RTS
                                         ;
                                         ; Enter BYT1DL with the delay value in .X.
C00E  CA          BYT1DL   DEX           ; decrement the one-byte delay value
C00F  D0 FD                BNE  BYT1DL   ; if .X is greater than zero, continue
C011  60                   RTS           ; we're finished
```

See also BYT2DL, INTDEL, JIFDEL, KEYDEL, TOD1DL.

Name

Cause a two-byte delay

Description

Like **BYT1DL**, this routine also produces short program delays. But with **BYT2DL**, the delays are slightly longer—from a few milliseconds (1/1000 second) to roughly 1/3 second. Delays on this order are frequently needed in writing game programs, especially when you move sprites about the screen.

Prototype

1. Enter this routine with the delay byte in .X.
2. Initialize .Y to zero. Then in YLOOP, decrement .Y until it reaches zero (256 times).
3. When .Y reaches zero, decrement .X, repeating YLOOP each time until .X reaches zero. Then RTS to the main program.

Explanation

To use **BYT2DL**, load the X register with a delay byte and JSR to the routine. Notice that because of the BNE instruction in $C014, a maximum delay actually occurs when the incoming value of .X is zero. In this case, the loop from $C00E to $C015 repeats 265 times.

As with **BYT1DL**, the actual amount of time that elapses during a delay, based on the initial value of .X, can be calculated from the number of machine cycles in the routine. (See that entry for an explanation of the calculation method used.) If we assume no page boundaries are crossed, each time YLOOP executes, it requires 5 * 256 - 1, or 1279, cycles. Each cycle takes approximately a millionth of a second.

The remaining instructions are the LDY at $C00E (2 cycles), the DEX at $C013 (2 cycles), the BNE at $C014 (3 cycles), the RTS at $C016 (6 cycles), and a JSR to the routine (6 cycles). Again, assuming no page boundaries are crossed, the number of machine cycles (MC) for the entire routine can be determined using the equation $MC = (1279 + 2 + 2 + 3) * X - 1 + 12$.

The X here represents the number of times the loop repeats. In all cases but one, X in the formula is the same as the X register. If .X is initially zero, use 256 for X in the formula.

Based on the clock speed for the 64 or 128, and with X varying from 0 to 256 in the formula, each delay can take from $1286 * 1 - 1 + 12 = 1297$ cycles to $1286 * 256 - 1 + 12 = 329{,}227$ cycles, or from 1.262 milliseconds to 0.322

seconds for North American (NTSC) systems.

Again, as with **BYT1DL**, additional instructions, such as NOPs, can be inserted into the code to adjust the delay times upward. In fact, using this approach, delays of a second or more can be achieved.

Routine

```
C000                 BGCOL0  =    53281      ; screen background color
C000                 DELAY   =    255        ; two-byte delay value
                                             ;
                                             ; Set the screen background color to light
                                             ; gray, cause a two-byte delay
                                             ; based on .X, and then change the
                                             ; background color to black.
C000  A9 0F          MAIN    LDA  #15        ; for light gray background
C002  8D 21 D0       BCKCOL  STA  BGCOL0
C005  A2 FF                  LDX  #DELAY     ; enter BYT2DL with the delay value in .X
C007  20 0E C0               JSR  BYT2DL     ; cause a delay
C00A  EE 21 D0               INC  BGCOL0     ; to produce a black background (only low
                                             ; nibble is significant)
C00D  60                     RTS
                                             ;
                                             ; Enter BYT2DL with the delay value in .X
C00E  A0 00          BYT2DL  LDY  #0         ; initialize .Y
C010  88             YLOOP   DEY
C011  D0 FD                  BNE  YLOOP      ; .Y decrements 256 times
C013  CA                     DEX             ; now decrement the delay value in .X
C014  D0 F8                  BNE  BYT2DL     ; continue if .X is greater than 0
C016  60                     RTS             ; we're finished
```

See also BYT1DL, INTDEL, JIFDEL, KEYDEL, TOD1DL.

Name

Print a one-byte integer

Description

At some point, programs that handle numbers—such as games, financial programs, and scientific and mathematical programs—are bound to require a routine that prints a one-byte integer. Not only is a routine like **BYTASC** ideal in programs of this type, but it can also be handy in overall program debugging.

For instance, suppose you have a problem in a lengthy section of coding. Knowing that **BYTASC** prints the one-byte value in the accumulator, you may be able to isolate your problem by transferring certain intermediate values to .A and JSRing to **BYTASC**.

Prototype

1. Enter this routine with the one-byte integer you wish to print in .A.
2. Initialize a place-holder table by storing three ASCII zeros—CHR$(48)—to it.
3. Set up a table of subtrahends for each digit's place—100, 10, 1.
4. Count the number of times (beginning with 48) the subtrahend representing the largest digit's place (100) can be subtracted from the value in the accumulator before a number less than zero results.
5. Store this number to the proper position in the place-holder table.
6. Repeat steps 4 and 5 for the next two digit places—10 and 1.
7. Finally, print out the ASCII place-holder table.

Explanation

In the example program, we fetch a one-byte value from the jiffy clock and print it with **BYTASC**.

The integer occupying any single-byte location is necessarily confined to a range from 0–255. This number can have as many as three digits when it's printed as a decimal number.

With this in mind, we set up a counter table (DIGITS—see below) containing three ASCII zeros, or CHR$(48)s. A common subtraction technique is then employed to convert the single-byte value in the accumulator into an equivalent string.

In this method, begin with the highest digit for the number, or the 100's place. We repeatedly subtract 100—the first entry in the table of one-byte subtrahends, or TB1SUB—from the number until a negative result occurs. After each subtraction yielding a positive value (>=0), increment the first entry in the DIGITS table representing the 100's place. When subtraction finally gives a negative value, the number is restored to the value it had before this last subtraction, and the whole process is repeated for the next two digits (the 10's place, then the 1's place).

When all three places in the number have been accounted for, DIGITS contains a three-byte string for the number. This is printed out beginning at DONE.

By maintaining a flag (ZEROFL) within the printing routine, we're able to print the number without printing any leading zeros. The flag tells us whether a nonzero digit has been printed. It has a value of zero as long as the preceding digits are all zeros. Whenever the first nonzero digit is encountered by the routine, ZEROFL takes on this value. In other words, it's no longer zero.

The printing routine must also consider the special case where the byte has a value of zero (all three digits are zero). This is taken care of within OUT. If ZEROFL is still zero after all three digits have been assessed, we print a zero.

Routine

```
C000              CHROUT  =    65490
C000              GETIN   =    65508
C000              JIFFY   =    162
C000              ZP      =    251
                                         ;
C000  A5 A2       LOOP    LDA  JIFFY     ; get a jiffy
C002  20 15 C0            JSR  BYTASC    ; convert value to ASCII and print it
C005  A9 20               LDA  #32       ; print a SPACE
C007  20 D2 FF            JSR  CHROUT
C00A  A9 0D               LDA  #13       ; print a RETURN
C00C  20 D2 FF            JSR  CHROUT
C00F  20 E4 FF            JSR  GETIN     ; check for keypress
C012  F0 EC               BEQ  LOOP      ; if no key, continue
C014  60                  RTS
                                         ;
                                         ; BYTASC converts the one-byte number in .A
                                         ; to ASCII and prints it.
C015  A2 30       BYTASC  LDX  #48       ; initialize place-holder table (DIGITS) with
                                         ; ASCII 0
C017  8E 63 C0            STX  DIGITS
C01A  8E 64 C0            STX  DIGITS+1
C01D  8E 65 C0            STX  DIGITS+2
C020  A0 00               LDY  #0        ; as an index
C022  8C 6A C0            STY  ZEROFL    ; initialize ZEROFL
C025  BE 63 C0    NMLOOP  LDX  DIGITS,Y  ; load with ASCII counter for a particular
                                         ; digit's place
```

```
C028  F0 14            BEQ   DONE       ; if we've reached the last digit's place, go
                                        ; print the number
C02A  38               SEC
C02B  F9 67 C0 SUBLOP  SBC   TB1SUB,Y   ; subtract corresponding table value from .A
C02E  E8               INX              ; increment ASCII counter for a particular
                                        ; digit's place
C02F  B0 FA            BCS   SUBLOP     ; if .A is still zero or above
C031  79 67 C0         ADC   TB1SUB,Y   ; we subtracted one time too many, so add
                                        ; subtrahend back to .A
C034  CA               DEX              ; since one time too many
C035  48               PHA              ; temporarily save .A
C036  8A               TXA
C037  99 63 C0         STA   DIGITS,Y   ; store respective digit to place-holder table
C03A  68               PLA              ; restore .A
C03B  C8               INY              ; for next digit's place
C03C  D0 E7            BNE   NMLOOP     ; branch always
C03E  A0 FF    DONE    LDY   #255       ; as index in the number
C040  C8       PRTLOP  INY              ; start with first digit
C041  B9 63 C0         LDA   DIGITS,Y
C044  F0 12            BEQ   OUT        ; if we're at the end of the table, leave routine
C046  AE 6A C0         LDX   ZEROFL     ; check ZEROFL to see if a nonzero digit has
                                        ; been printed
C049  D0 07            BNE   PRINT      ; if so, go print the digit
C04B  C9 30            CMP   #48        ; check for leading zeros
C04D  F0 F1            BEQ   PRTLOP     ; if leading zero occurs, get the next digit
C04F  8D 6A C0         STA   ZEROFL     ; store nonzero digit
C052  20 D2 FF PRINT   JSR   CHROUT     ; print each digit
C055  4C 40 C0         JMP   PRTLOP     ; and go to next place
C058  AD 6A C0 OUT     LDA   ZEROFL     ; determine if the number is 000
C05B  D0 05            BNE   EXIT       ; if not, then return
C05D  A9 30            LDA   #48        ; print a zero
C05F  20 D2 FF         JSR   CHROUT
C062  60       EXIT    RTS              ; we're finished
                                        ;
C063  00 00 00 DIGITS  .BYTE 0,0,0      ; for storing ASCII counter values for each
                                        ; digit's place
C066  00               .BYTE 0          ; digit's terminator byte
C067  64 0A 01 TB1SUB  .BYTE 100,10,1   ; table of one-byte subtrahends for each digit's
                                        ; place
C06A  00       ZEROFL  .BYTE 0          ; ZEROFL is nonzero if a nonzero digit has
                                        ; printed
```

See also CNUMOT, FACPRD, FACPRT, NUMOUT.

Name

Convert an ASCII number to a binary integer value

Description

The four characters in the string *1025* translate to the hex value $0401. If you have a program in which you expect users to type in individual numbers such as *1, 0, 2,* and *5,* and if you'd like to change the characters to a workable integer, this routine will handle the conversion.

Prototype

1. Zero the bytes that hold the result.
2. Get the first (or next) character and subtract 48 to strip off the ASCII trappings.
3. Multiply the result by 10 and add the new value.
4. Jump back to step 2 and get the next character; repeat until all have been taken care of.

Explanation

This example routine can take nine or ten ASCII characters and translate them into binary values. The limit is the four bytes of the RESULT variable; four bytes can count up to approximately 4.3 billion (4,294,967,206). It should be relatively easy to add a fifth byte (or even more) to extend the range to the size of the U.S. budget.

The **CAS2IN** routine has no error checking. It's up to you to make sure the characters are within the range 48–57 (ASCII 0–9). The ASCII string should be terminated with a zero byte, or with any byte that's less than 48, for that matter.

An example of conversion is the short string *9801,* which contains four characters. Start at the leftmost character, *9.* Multiply the result (0) by 10 (still 0) and add 9. Now the result holds a 9. The next character is *8.* Multiply the result (9) by 10 (90) and add 8 (98). The next character is *0.* Multiply result (98) by 10 (980) and add 0 (980). The final character is *1.* So, 980 becomes 9800, then 9801. The ASCII string of characters *9801* (the four characters $39, $38, $30, and $31) has been transformed into the numeric value $2649.

One of the key routines is TIMES10. If you shift a binary number to the left, you multiply it by 2. Likewise, if you shift a decimal number to the left, you multiply by 10. For example, 120 shifted left in base 10 is 1200 (120 × 10). In binary, %1101 (decimal 13) shifted left becomes %11010 (decimal 26). To multiply a binary number by 10, shift it left once (times 2)

and save it. Then shift left two more times (times 2 times 2, for a grand total of times 8). Add the two results (x times 2 plus x times 8) and the number has been multiplied by 10.

Warning: Don't succumb to the temptation to replace the multiple ADC or ROL instructions with an indexed loop. The X and Y registers would have to count from zero to three, because the low byte comes before medium and high bytes, and you have to add or rotate the low byte first. To test for the end of the loop, you'd have to CPX or CPY. But the act of comparing sets or clears the carry flag, and the whole point of the addition or rotation is to move the carry flags as they overflow from one byte to the next. If you compare, you change the carry flag, with potentially weird results.

Routine

```
C000                CAS2IN   =     *
C000   A9  00                LDA   #0          ; first, zero out the total
C002   A2  03                LDX   #BYTES      ; number of bytes in the result
C004   9D  78  C0   ZLOOP    STA   RESULT,X
C007   CA                    DEX               ; count down
C008   10  FA                BPL   ZLOOP       ; and loop
C00A   AA                    TAX               ; .A holds a zero
C00B   BD  71  C0   MAINLP   LDA   ASCNUM,X    ; get a number
C00E   38                    SEC               ; strip off the ASCII part to get a number
                                               ; 0-9
C00F   E9  30                SBC   #48         ; by subtracting 48
C011   90  1E                BCC   FINIS       ; If the number is less than 48, carry is
                                               ; clear
C013   48                    PHA               ; save this number temporarily
C014   20  32  C0            JSR   TIMES10     ; multiply RESULT by 10
C017   68                    PLA               ; get the value again
C018   18                    CLC
C019   6D  78  C0            ADC   RESULT      ; and add it to RESULT
C01C   8D  78  C0            STA   RESULT      ; store it back
C01F   90  0D                BCC   DOX
C021   EE  79  C0            INC   RESULT+1    ; do the high bytes
C024   D0  08                BNE   DOX
C026   EE  7A  C0            INC   RESULT+2
C029   D0  03                BNE   DOX
C02B   EE  7B  C0            INC   RESULT+3
C02E   E8           DOX      INX               ; count forward
C02F   D0  DA                BNE   MAINLP      ; and go back for more/branch always
                                               ;
C031   60           FINIS    RTS               ; end of the routine
                                               ;
C032   20  42  C0   TIMES10  JSR   TIMES2      ; multiply RESULT by 2
C035   20  4F  C0            JSR   COPYIT      ; copy RESULT to TEMP
C038   20  3F  C0            JSR   TIMES4      ; multiply RESULT by 4 more (total of 8)
C03B   20  5B  C0            JSR   ADDEM       ; add TEMP and RESULT (times 10 total)
C03E   60                    RTS               ; done
                                               ;
C03F   20  42  C0   TIMES4   JSR   TIMES2      ; call TIMES2 and fall through
C042   0E  78  C0   TIMES2   ASL   RESULT      ; times 2 via shifts to the left
C045   2E  79  C0            ROL   RESULT+1
C048   2E  7A  C0            ROL   RESULT+2
C04B   2E  7B  C0            ROL   RESULT+3
```

```
C04E  60                    RTS                   ; end of times routines
                                                  ;
C04F  A0  03      COPYIT    LDY   #BYTES          ; copy
C051  B9  78  C0  CPLOOP    LDA   RESULT,Y        ; from RESULT
C054  99  7C  C0            STA   TEMP,Y          ; to TEMP
C057  88                    DEY
C058  10  F7                BPL   CPLOOP          ; branch back
C05A  60                    RTS
                                                  ;
C05B  A0  00      ADDEM     LDY   #0
C05D  18                    CLC
C05E  08                    PHP                   ; preparation
C05F  28          ADLOOP    PLP                   ; get .P back
C060  B9  7C  C0            LDA   TEMP,Y          ; get TEMP
C063  79  78  C0            ADC   RESULT,Y        ; add it to RESULT
C066  99  78  C0            STA   RESULT,Y        ; and store it
C069  08                    PHP                   ; save carry temporarily
C06A  C8                    INY
C06B  C0  04                CPY   #MAX
C06D  D0  F0                BNE   ADLOOP
C06F  28                    PLP                   ; remove .P from the stack
C070  60                    RTS
                                                  ;
C071  31  39  36  ASCNUM    .ASC  "196863"
C077  00                    .BYTE 0               ; always zero terminated
C078  00  00  00  RESULT    .BYTE 0,0,0,0         ; enough to handle roughly 4,000,000,000
                                                  ; but you can add more zeros for larger
                                                  ; numbers
C07C              MAX       =     *   RESULT
C07C              BYTES     =     MAX - 1
C07C  00  00  00  TEMP      .BYTE 0,0,0,0
```

See also BCD2AX, CB2ASC, CB2HEX, CI2HEX.

Name

Convert Commodore ASCII characters into screen codes

Description

Both the 64 and 128 represent their character sets in different ways, depending on the application. **CASSCR** converts characters from one of these coding systems to another—namely, from Commodore ASCII into screen codes. This routine is helpful anytime you need to store Commodore ASCII characters or strings of characters directly into screen memory. Two popular word processors (*WordPro* and *SpeedScript*) store their files as screen codes, so this routine is useful in performing conversions of ASCII files to be used with these word processors.

The routine itself is set up to receive Commodore ASCII character values in the accumulator. An equivalent screen code, if it exists, is then returned in the accumulator. In the process, the carry flag is cleared. However, if no screen code is defined for the character, the accumulator is left unchanged, and carry is set to indicate the conversion error.

Prototype

1. Check for the pi character (255). If the character is pi, set .A to 126, clear carry, and return.
2. Otherwise, determine whether the character lacks an equivalent screen code value (character code is in the range 0–31 or 128–159). If so, set the carry flag and return, leaving .A as is.
3. If the character's value exceeds 127, go to step 5.
4. If it's in the range 96–127, AND it with 95, clear carry, and return.
5. Replace bit 6 of .A with bit 7, place a zero in bit 7, and RTS, leaving carry clear.

Explanation

The example program converts a string of Commodore ASCII characters (in STRING) into screen codes and stores them at the beginning of screen memory. Any characters that lack screen codes won't appear (BCS SKIP).

Except for the special case of character 255, which is set to 126, **CASSCR** performs conversions based on the range in which the character lies. As it turns out, each range can be characterized by a different bit pattern. The table shows the bit patterns of characters within each range before and after conversion.

Character Bit Patterns

	Before:		After:
Range	**Bit Pattern**	**Range**	**Bit Pattern**
0–31	%000*x xxxx*	Nonexistent	
128–159	%100*x xxxx*	Nonexistent	
96–127	%011*x xxxx*	64–95	%010*x xxxx*
32–63	%001*x xxxx*	32–63	%001*x xxxx* (same)
64 95	%010*x xxxx*	0–31	%000*x xxxx*
160–191	%101*x xxxx*	96–127	%011*x xxxx*
192–223	%110*x xxxx*	64–95	%010*x xxxx*
224–254	%111*x xxxx*	96–127	%011*x xxxx*

Within each bit pattern, a zero designates bits that are always off; a one designates bits that are always on. An *x* represents bits that may be on or off.

We've intentionally grouped together character ranges that can be converted with the same bit manipulations. The first group is handled as in step 2 of the prototype above, the second as in step 4, and the third as in step 5.

If you look closely at the bit patterns, you'll see how the routine will work. First, if the result of the number AND 127 (%01111111) is 31 or less, the ASCII value can't be converted. If the number is in the range 96–127, AND it with 95 (%01011111), and you're finished. The final and largest group has three characteristics: Bit 7 is always %0 in the result. Bit 6 of the screen code is always the same as bit 7 of the ASCII code. And bit 5 remains unchanged.

The overall effect is that ASCII characters without screen codes (in the range 0–31 or 128–159) are left alone, but the carry flag is set. For all others, the carry flag is cleared.

Note: **CASSCR** has no effect on either .X or .Y. For this reason, you can use the routine in a loop indexed by either register without first having to save the register contents.

Routine

```
C000        CHROUT  =   65490
C000        ZP      =   251
C000        SCREEN  =   1024    ; start of text screen
C000        COLRAM  =   55296   ; start of color RAM
C000        BGCOL0  =   53281   ; screen background color
C000        BLACK   =   0
C000        MDGRAY  =   12
```

```
                                                ;
                                                ; Convert a string from Commodore ASCII to
                                                ; screen codes and POKE it.
C000  A9 93     CLRCHR  LDA  #147               ; clear the screen
C002  20 D2 FF          JSR  CHROUT
C005  A9 0C             LDA  #MDGRAY            ; set screen background color to medium gray
C007  8D 21 D0          STA  BGCOL0
C00A  A0 00             LDY  #0                 ; as an index
C00C  B9 5A C0  LOOP    LDA  STRING,Y           ; get a character from string
C00F  F0 10             BEQ  FINISH             ; is it a zero byte?
C011  20 22 C0          JSR  CASSCR             ; convert it to a screen code
C014  B0 08             BCS  SKIP               ; if carry is set, no screen code exists
C016  99 00 04          STA  SCREEN,Y           ; POKE message to screen using modified
                                                ; POKSCR
C019  A9 00             LDA  #BLACK             ; set foreground color of character to black (for
                                                ; early 64s)
C01B  99 00 D8          STA  COLRAM,Y
C01E  C8        SKIP    INY                     ; next character
C01F  D0 EB             BNE  LOOP               ; continue printing
C021  60        FINISH  RTS
                                                ;
                                                ; Convert Commodore ASCII in .A to screen
                                                ; code in .A.
                                                ; If no corresponding screen code exists, carry
                                                ; is set to indicate the error and
                                                ; .A is unchanged.
C022  C9 FF     CASSCR  CMP  #255               ; is it pi?
C024  D0 04             BNE  NEQUIV             ; if not, check for nonequivalent codes
C026  A9 7E             LDA  #126               ; 255 becomes 126
C028  18                CLC
C029  60                RTS                     ; and we exit
C02A  8D 80 C0  NEQUIV  STA  TEMPA              ; preserve Commodore ASCII value for later
                                                ; checks
C02D  29 60             AND  #%01100000         ; check for nonequivalent codes (0-31 and
                                                ; 128-159)
C02F  D0 05             BNE  UPPLOW             ; if no, go check for upper/lower half of
                                                ; character set
C031  AD 80 C0  ERROR   LDA  TEMPA              ; otherwise, no equivalent code so restore .A
C034  38                SEC                     ; and indicate error
C035  60                RTS
C036  AD 80 C0  UPPLOW  LDA  TEMPA              ; restore .A
C039  30 06             BMI  REMAIN
C03B  29 60             AND  #%01100000         ; in lower half; first check whether in range
                                                ; 96-127
C03D  C9 60             CMP  #%01100000         ; bit 5 and 6 are set if in 96-127
C03F  F0 12             BEQ  TOPLOW             ; if so, go convert
                                                ;
                                                ; Otherwise, handle remainder (32-63, 64-95,
                                                ; 160-191, 192-223, 224-254).
                                                ; Shift bit 7 to 6 of TEMPA (containing the
                                                ; character) and set bit 7 to 0.
C041  0E 80 C0  REMAIN  ASL  TEMPA              ; bit 7 of TEMPA into carry
C044  2A                ROL                     ; carry into bit 0 of .A
C045  2E 80 C0          ROL  TEMPA              ; bit 6 of original TEMPA goes into carry
C048  6A                ROR                     ; bit 0 of .A back into carry
C049  6E 80 C0          ROR  TEMPA              ; carry into bit 7
C04C  4E 80 C0          LSR  TEMPA              ; move 7 to 6 while setting 7 to 0
C04F  AD 80 C0          LDA  TEMPA              ; restore .A
C052  60                RTS                     ; and return (the LSR cleared the carry)
C053  AD 80 C0  TOPLOW  LDA  TEMPA              ; convert range 96-127
C056  29 5F             AND  #%01011111
C058  18                CLC                     ; and return with an equivalent code
C059  60                RTS
```

```
                                        ;
C05A  54  C8  C9  STRING   .ASC         ; "This message was POKEd to the screen."
C07F  00                   .BYTE0
C080              TEMPA    .BYTE0       ; for temporary .A storage
```

See also CASTAS, CNVERT, SCRCAS, TASCAS.

Name

Convert Commodore ASCII characters to true ASCII

Description

Commodore computers, including the 64 and 128, use their own special character codes known as Commodore ASCII. Many other microcomputers use a more standard character set known as true ASCII. On a 64 or 128, for example, the ASCII character 65 is a lowercase *a*. But true ASCII defines 65 as an uppercase *A*. This is the primary difference between the two ASCIIs: The upper and lowercase letters are switched. In order to send transmissions via a modem to other computers, or to use certain printers that expect to receive true ASCII, you need to convert Commodore's ASCII to true ASCII.

CASTAS converts Commodore characters in the accumulator to true ASCII and leaves the result in .A. All true ASCII characters are in the range 0–127. Ordinarily, no characters above 127, most of which are graphics characters, will be converted. However, the 64 and 128 have a second set of uppercase characters, 193–218, which are used when printing to the screen. In addition, shifted-space—CHR$(160)—is sometimes typed in as if it were a normal space (when SHIFT LOCK is engaged, for example). So these two instances are exceptions to the rule.

Also, there are characters on the 64 and 128 for which there are no true ASCII equivalents. If **CASTAS** receives one of these, it returns a zero in the accumulator and sets the carry flag.

Prototype

1. Change the shifted-space character (160), if it occurs, to space (32).
2. Check the character value to see whether it lies within one of three ranges of Commodore ASCII alphabetic characters (193–218, 97–122, or 65–90).
3. If it doesn't, go to step 7.
4. If the character in .A is within one of the three ranges, ASL it.
5. If carry is clear (so the character is either in the range 97–122 or 65–90), flip bit 6. Otherwise, go to step 6 (the character is 193–218).
6. Perform an LSR.
7. Determine whether the character value is 128 or greater. If it's not, then RTS.
8. Otherwise, set .A to zero and leave carry set.

Explanation

You can test this routine in the example program by typing in all sorts of Commodore ASCII characters. As each character is typed in, its Commodore ASCII value is displayed, conversion is done with **CASTAS**, and the equivalent true ASCII value is also shown. This process continues until you press RETURN.

Conversion from Commodore ASCII to true ASCII is fairly straightforward because of the similarities between the two character sets. Basically, all we need to do is switch uppercase (97–122 or 193–218) to lowercase (65–90) letters, or lowercase to uppercase (97–122). This is all handled by the routine **SWITCH**, explained elsewhere in this book. As mentioned, the shifted-space is a special case. So, before entering **SWITCH**, we convert this character to a normal space (32).

Note: **CASTAS** corrupts the .Y register. If your program uses .Y, be sure to save it to a temporary location before entering the routine. And, of course, restore it when you return from **CASTAS**.

Routine

```
C000                CHROUT  =    65490
C000                GETIN   =    65508
C000                LINPRT  =    48589         ; LINPRT = 36402 on the 128
C000                ZP      =    251
C000                DSFTCM  =    8             ; DSFTCM = 11 on the 128
C000                ESFTCM  =    9             ; ESFTCM = 12 on the 128
                                               ;
                                               ; Get a character; print its Commodore ASCII
                                               ; value and true ASCII value.
                                               ; Quit on RETURN
C000  A9 0E                 LDA  #14           ; switch to lowercase/uppercase mode
C002  20 D2 FF              JSR  CHROUT
C005  A9 08                 LDA  #DSFTCM       ; disable SHIFT/Commodore key case
                                               ; switching
C007  20 D2 FF              JSR  CHROUT
C00A  20 E4 FF  WAIT        JSR  GETIN         ; get a character
C00D  F0 FB                 BEQ  WAIT          ; if null string, then get another key
C00F  20 30 C0              JSR  NUMPRT        ; print the Commodore ASCII value
C012  A9 20                 LDA  #32           ; print space
C014  20 D2 FF              JSR  CHROUT
C017  A5 FB                 LDA  ZP            ; restore .A
C019  20 38 C0              JSR  CASTAS        ; convert value in .A from Commodore to
                                               ; true ASCII
C01C  20 30 C0              JSR  NUMPRT        ; print the true ASCII value
C01F  A9 0D                 LDA  #13           ; print RETURN
C021  20 D2 FF              JSR  CHROUT
C024  A5 FB                 LDA  ZP            ; restore .A
C026  C9 0D                 CMP  #13           ; is it RETURN?
C028  D0 E0                 BNE  WAIT          ; no, so get another character
C02A  A9 09     QUIT        LDA  #ESFTCM       ; enable SHIFT/Commodore key case
                                               ; switching
C02C  20 D2 FF              JSR  CHROUT
C02F  60                    RTS                ; and return
                                               ;
```

```
C030  85 FB       NUMPRT   STA   ZP          ; save .A
C032  AA                   TAX               ; low byte of ASCII value
C033  A9 00                LDA   #0          ; high byte
C035  4C CD BD    NUMOUT   JMP   LINPRT      ; print the ASCII value (see NUMOUT) and
                                             ; return
                                             ;
                                             ; Convert Commodore ASCII in .A to true
                                             ; ASCII in .A.
                                             ; .A is zero and carry flag is set if there is no
                                             ; equivalent true ASCII value
                                             ; (except characters 193-218, which are
                                             ; converted as if they were 97-122).
                                             ; Also character 160 is handled as if it were a
                                             ; 32
C038  C9 A0       CASTAS   CMP   #160        ; take care of shift-space
C03A  D0 02                BNE   SWITCH      ; if not shift-space, use SWITCH to convert
                                             ; others
C03C  A9 20                LDA   #32         ; shift-space becomes space
C03E  A0 03       SWITCH   LDY   #3          ; index to table
C040  88          LOOP     DEY
C041  30 10                BMI   EXIT        ; exit if no more ranges to check
C043  D9 5A C0             CMP   RANGE1,Y
C046  90 0B                BCC   EXIT        ; character falls below RANGE1, so exit
C048  D9 5D C0             CMP   RANGE2,Y
C04B  B0 F3                BCS   LOOP        ; character is above RANGE2 so check next
                                             ; range
C04D  0A          FLIPIT   ASL               ; character is in a range; shift bit 7 to carry
C04E  B0 02                BCS   FIXIT       ; character is >=128
C050  49 40                EOR   #64         ; flip bit 6
C052  4A          FIXIT    LSR               ; restore character (bit 7 becomes zero)
C053  C9 80       EXIT     CMP   #128        ; carry is set for all characters above 128
                                             ; (except 193-218 and 160)
C055  90 02                BCC   OUT
C057  A9 00                LDA   #0          ; return a zero in .A if above 128 (and not
                                             ; exceptions)
C059  60          OUT      RTS
                                             ;
C05A  C1 61 41    RANGE1   .BYTE 193,97,65   ; lower delimiter of each range
C05D  DB 7B 5B    RANGE2   .BYTE 219,123,91  ; upper delimiter+1 of each range
```

See also CASSCR, CNVERT, SCRCAS, TASCAS.

Name

Convert a byte value to an ASCII number by using subtraction

Description

A byte value such as 102 is stored in memory as a series of binary bits. If you want to print it out, not as a CHR$(102), but as the three characters 1, 0, and 2, you can use this routine to convert the byte to three ASCII values.

Prototype

1. Enter **CB2ASC** with the value to be translated in the accumulator.
2. Load .Y with a zero.
3. Repeatedly subtract 100 until the value becomes negative.
4. After each successful subtraction, increment .Y.
5. When the value becomes less than zero, add back 100 and store .Y.
6. Repeat steps 2–5, subtracting the values 10 and 1.

Explanation

The procedure is straightforward. Subtract hundreds, then tens, then ones. At each step, save the result in memory. These numbers can then be ORed with 48 to create printable ASCII numbers.

Routine

```
C000                 TIMER    =    $A2          ; the jiffy clock
C000                 RESULT   =    828          ; store ASCII digits in the cassette buffer
                                                ; (RESULT = 2816 on the 128)
C000                 CHROUT   =    $FFD2
                                                ;
C000  A5 A2                   LDA  TIMER        ; get a changing number
C002  20 15 C0                JSR  CB2ASC       ; convert it
C005  A0 00                   LDY  #0           ; loop counter
C007  B9 3C 03       LOOP     LDA  RESULT,Y     ; get the ASCII numbers one by one
C00A  09 30                   ORA  #%00110000   ; make it ASCII
C00C  20 D2 FF                JSR  CHROUT       ; print it
C00F  C8                      INY               ; counter increases
C010  C0 03                   CPY  #3           ; quit after 0-2
C012  D0 F3                   BNE  LOOP         ; or go back
C014  60                      RTS               ; end of the framing routine
                                                ;
C015                 CB2ASC   =    *
C015  A0 00                   LDY  #0           ; .Y is the counter
C017  38                      SEC               ; get ready to subtract
C018  E9 64          HLOOP    SBC  #100         ; keep subtracting
C01A  90 03                   BCC  TENS         ; until we've gone past zero
C01C  C8                      INY               ; count up by one
C01D  D0 F9                   BNE  HLOOP        ; and loop back
                                                ;
C01F  8C 3C 03       TENS     STY  RESULT       ; .Y holds the hundred's place
C022  A0 00                   LDY  #0           ; zero it again
```

```
C024  69  64                ADC  #100         ; set .A back to normal
C026  38                    SEC
C027  E9  0A       TLOOP    SBC  #10          ; this time, minus 10
C029  90  03                BCC  ONES         ; carry clear means underflow
C02B  C8                    INY               ; else, inc the counter
C02C  D0  F9                BNE  TLOOP        ; and go back to subtract
                                              ;
C02E  8C  3D  03   ONES     STY  RESULT+1     ; .Y is the ten's place
C031  69  0A                ADC  #10          ; add 10 to .A
C033  8D  3E  03            STA  RESULT+2     ; and store it
C036  60                    RTS               ; end of routine
```

See also BCD2AX, CAS2IN, CB2HEX, CI2HEX.

Name

Convert a byte value (0–99) to a BCD number

Description

Bytes range in value from 0 to 255 ($00 to $FF). BCD numbers, on the other hand, can only have 100 values ($00–$99). This routine converts a byte in the range 0–99 decimal to a BCD value.

Prototype

1. Isolate the high nybble.
2. Compare to 10. If the high nybble is more than 10, subtract 10.
3. Rotate the carry flag into ANSWER.
4. Loop back to step 2 five times.
5. Shift ANSWER to the left and OR the remainder in .A with ANSWER.

Explanation

The framing routine takes a value from one location ($FB), calls the conversion routine, and stores the result in a second location ($FC). Note that at the beginning of **CB2BCD**, numbers greater than 99 are trapped by subtracting 100 until the value is in the proper range. This means that if you enter with a value of 132, the result will be $32.

Routine

```
C000  A5 FB        MAIN     LDA  $FB       ; get a byte from memory
C002  20 08 C0              JSR  CB2BCD    ; convert it
C005  85 FC                 STA  $FC
C007  60                    RTS            ; end of routine
                                           ;
C008               CB2BCD   =    *
C008  C9 64        PRELIM   CMP  #100      ; first check the range
C00A  90 04                 BCC  BEGIN     ; ready to start if it's 0-99
C00C  E9 64                 SBC  #100      ; subtract 100
                                           ; Put an INC here if you want.
C00E  B0 F8                 BCS  PRELIM    ; branch always
                                           ;
C010  8D 45 C0     BEGIN    STA  TEMPA     ; store it
C013  A9 00                 LDA  #0        ; ready to ROL
C015  8D 46 C0              STA  ANSWER    ; the answer will be here
C018  A0 04                 LDY  #4        ; four times
C01A  0E 45 C0     RLOOP    ASL  TEMPA     ; move the high bit into carry
C01D  2A                    ROL            ; and rotate it into .A
C01E  88                    DEY            ; count down
C01F  D0 F9                 BNE  RLOOP     ; four times
                                           ;
C021  A0 05                 LDY  #5        ; this loop happens five times
C023  C9 0A        CLOOP    CMP  #10       ; is .A bigger than 10?
C025  08                    PHP            ; save the status
C026  2E 46 C0              ROL  ANSWER    ; and put carry into answer
C029  28                    PLP            ; get .P back
```

```
C02A  90  02               BCC   AHEAD     ; if clear, leave .A alone
C02C  E9  0A               SBC   #10       ; else subtract 10
C02E  0E  45  C0  AHEAD    ASL   TEMPA     ; shift left
C031  2A                   ROL             ; into .A
C032  88                   DEY             ; loop
C033  D0  EE               BNE   CLOOP     ; back to the compare
                                           ; .A contains the remainder.
C035  4A                   LSR             ; .A needs to be corrected
C036  A0  04               LDY   #4        ; four shifts
C038  0E  46  C0  AALOOP   ASL   ANSWER
C03B  88                   DEY
C03C  D0  FA               BNE   AALOOP
C03E  0D  46  C0           ORA   ANSWER    ; add the low nybble
C041  8D  46  C0           STA   ANSWER
C044  60                   RTS
                                           ;
C045  00          TEMPA    .BYTE 0
C046  00          ANSWER   .BYTE 0
```

See also B2SNIN, B2UNIN, BCD2BY, CFP2I, CI2FP, CNVBFP.

Name

Convert a byte to two hexadecimal digits (ASCII)

Description

When you're looking at the contents of memory, hexadecimal (base 16) is sometimes preferable to decimal or binary. **CB2HEX** takes a single number (in the range 0–255) as input and returns the two ASCII characters that make up the hexadecimal equivalent.

Prototype

1. Enter the routine with the value in .A.
2. Temporarily save it.
3. AND with the mask value %00001111 to extract the lower nybble and ORA with 48 ($30) to convert to ASCII. If the result is greater than 57 (ASCII 9), add 7 to put it in the range A–F. This result goes into .X.
4. Retrieve the original value and shift it right four times.
5. Repeat step 3 to convert the high nybble to an ASCII value.

Explanation

The example routine gets a keypress, checks for the letter *Q* (quit) and then prints four things: the letter pressed, the decimal ASCII value of the character for the key, the hexadecimal equivalent of the decimal number, and a RETURN. It then loops back to get another key.

The subroutine is fairly simple. It first extracts the low nybble and high nybble (a *byte* contains eight bits, while a *nybble* is half a byte—four bits). The nybbles are then converted to ASCII. Because the characters 0–9 correspond to the ASCII codes 48–57, and the characters A–F are ASCII 65–70, it's sometimes necessary to add 7 to bridge the gap between the character codes for 9 and A.

A few techniques bear mentioning. First, the RTS that ends the framing routine occurs very early in the program ($C009). Most of the time, the program branches over this instruction. There's no rule that says a routine must have an RTS as the last instruction. Second, within the **CB2HEX** routine itself, the ASCSUB subroutine is used twice. The first time, $C034 performs a JSR ASCSUB. The subroutine executes once and returns back to $C037. The second time, the program falls through to ASCSUB. This time, the RTS ends the **CB2HEX** routine. The first time, the RTS ends a subroutine within the **CB2HEX** subroutine; the second time, it ends

CB2HEX itself. Finally, the ADC #6 at $C041 doesn't add 6; it adds 7. The instruction above is a BCC (Branch if Carry Clear) around the ADC instruction, which means *ADd with Carry*. If the carry flag is set, adding 6 plus a carry of 1 is the same as adding 7.

Note: The value of .A is temporarily stored in .Y at $C031. If you're using the Y register as a counter or index, you may wish to substitute PHA/PLA (or STA/LDA) for the TAY/TYA combination.

Routine

```
C000                CHROUT  =    $FFD2
C000                GETIN   =    $FFE4
C000                LINPRT  =    $BDCD         ; LINPRT = $8E32 on the 128
C000  20  E4  FF    MAIN    JSR  GETIN         ; get a key
C003  F0  FB                BEQ  MAIN          ; loop back if no key
C005  C9  51                CMP  #"Q           ; if Q then quit
C007  D0  01                BNE  CONTIN        ; else continue
C009  60                    RTS
C00A  20  D2  FF    CONTIN  JSR  CHROUT        ; print the character
C00D  48                    PHA                ; push it on stack
C00E  AA                    TAX                ; low byte in .X
C00F  A9  20                LDA  #32           ; character code for space
C011  20  D2  FF            JSR  CHROUT        ; print it
C014  A9  00                LDA  #0            ; high byte for LINPRT
C016  20  CD  BD            JSR  LINPRT        ; print the decimal value
C019  A9  3D                LDA  #"=
C01B  20  D2  FF            JSR  CHROUT        ; print an equal sign
C01E  68                    PLA                ; get the original number
C01F  20  31  C0            JSR  CB2HEX        ; convert it to hex
C022  20  D2  FF            JSR  CHROUT        ; print high nybble
C025  8A                    TXA                ; .X into .A
C026  20  D2  FF            JSR  CHROUT        ; print that, too
C029  A9  0D        END     LDA  #13           ; carriage return
C02B  20  D2  FF            JSR  CHROUT
C02E  4C  00  C0            JMP  MAIN          ; go back for more
                                               ;
C031                CB2HEX  =    *
C031  A8                    TAY                ; save contents of .A in .Y
C032  29  0F                AND  #%00001111    ; extract low nybble
C034  20  3D  C0            JSR  ASCSUB        ; the conversion subroutine
C037  AA                    TAX                ; put the low nybble in .X
C038  98                    TYA                ; retrieve the original number
C039  4A                    LSR
C03A  4A                    LSR
C03B  4A                    LSR
C03C  4A                    LSR                ; shift right four times
                                               ; Now fall through to the ASCSUB routine.
C03D  C9  0A        ASCSUB  CMP  #10           ; is it 0-9?
C03F  90  02                BCC  ADD48         ; yes, branch forward and add 48
                                               ; (for ASCII)
C041  69  06                ADC  #6            ; this really adds 7 because carry is set
C043  69  30        [illegible] ADC  #48       ; add 48 to make 0-9 into 48-57 or 10-15
                                               ; into A-F
C045  60                    RTS                ; and that's that
```

See also BCD2AX, CAS2IN, CB2ASC, CI2HEX.

Name

Print semilarge (4 × 4) characters

Description

This routine looks up the character shape in ROM and prints it out (to screen or printer) as a large character that's four times the normal size.

Prototype

1. Set up a zero-page pointer to the character shape.
2. Read the eight bytes from character ROM; store them in memory.
3. Loop four times, once through each pair of bytes.
4. Rotate each byte to the left twice to get a number 0–15.
5. Look up the appropriate graphics character and print it.
6. The resulting printout has four graphics characters on four lines.

Explanation

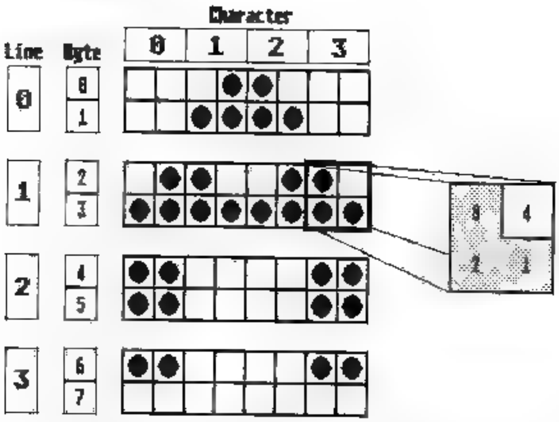
At the beginning of the routine, the screen code of the character to be printed is in the accumulator, and the choice of uppercase/graphics or lowercase/uppercase is determined by the carry flag. The first thing we have to do is find the character shape in ROM, so .A is stored in a free zero-page location, and a $0D is stored in the corresponding high byte of the pointer. A single ROL transfers the contents of the carry flag into FREEZP+1. Now we have either a $1A or $1B there. The next task is to multiply this two-byte pointer by 8, via three ASL/ROL pairs.

The pointer at $FB now points to the character shape, so we can look up the eight bytes that form the letter. The routine from $C014 through $C02B does this. The interrupts must first be turned off with SEI. Then bit 2 of location 1 is turned off so we can read character ROM. The shape is stored into CHCOPY (at the end of the program).

As an example, imagine that we're printing a large capital *A*. The figure shows how the bits are arranged in the character.

Start with a 0 in the accumulator. Rotate byte 0 to the left twice; then rotate byte 1 twice. The result is a number between 0 and 15 in the accumulator. This number is used as an offset to find out first whether we should print {RVS ON} or {RVS OFF}, and then which character to print. This procedure repeats four times, and we go down to the next row (bytes 2 and 3), and so on.

The Letter *A*



If you look at character 3 on line 1, you'll see that the graphics character to be printed should be a Commodore-C (in reverse mode).

Note: On the 64, it's necessary to turn off bit 2 of location 1 to get to the character set in ROM. On the 128, you can access the character by switching to bank 14. Thus, it's necessary to remove the instructions from SEI to STA $01 ($C014–$C01A) and the instructions from LDA $01 to CLI ($C025–$C02B). Also, the instruction LDA (FREEZP),Y at $C01D should be replaced with a call to the INDFET (indirect fetch) Kernal routine, as follows:

```
LOOP LDA #FREEZP  ; the zero page pointer
     LDX #14      ; the bank to access
     JSR 65396    ; the INDFET Kernal routine
```

Routine

```
C000                FREEZP   =    $FB
C000                CHROUT   =    $FFD2
                                                ; Enter with the screen code in .A.
                                                ; Carry clear for uppercase/graphics; carry set
                                                ; for lowercase/uppercase.
                                                ;
C000  85 FB  CHARX4   STA  FREEZP               ; screen code (low byte to multiply by 8)
C002  A9 0D           LDA  #%00001101           ; $0D, which will be shifted four times to
                                                ; become $D0 or $D8
C004  85 FC           STA  FREEZP+1             ; almost ready to rotate
C006  26 FC           ROL  FREEZP+1             ; got carry now
                                                ; Now multiply it by 8.
                                                ;
C008  06 FB           ASL  FREEZP
C00A  26 FC           ROL  FREEZP+1
C00C  06 FB           ASL  FREEZP
C00E  26 FC           ROL  FREEZP+1
C010  06 FB           ASL  FREEZP
```

```
C012  26  FC               ROL   FREEZP+1
                                              ; FREEZP now points to the first byte of
                                              ; character ROM.
                                              ;
C014  78                   SEI                ; turn off interrupts while we read
                                              ; character ROM
C015  A5  01               LDA   $01          ; bit 2 of location 1 controls character ROM
C017  29  FB               AND   #%11111011   ; mask it out to get to the characters
C019  85  01               STA   $01
C01B  A0  07               LDY   #7           ; need the eight bytes (0-7)
C01D  B1  FB       LOOP    LDA   (FREEZP),Y   ; get the shape
C01F  99  8B  C0           STA   CHCOPY,Y     ; and put it in memory
C022  88                   DEY                ; count down
C023  10  F8               BPL   LOOP         ; we want 0, so count down to $FF
C025  A5  01               LDA   $01          ; check location 1
C027  09  04               ORA   #%00000100   ; and turn the bit back on
C029  85  01               STA   $01
C02B  58                   CLI                ; interrupts are OK now
                                              ;
                                              ; Print the shape on the screen.
C02C  A9  04               LDA   #4
C02E  8D  93  C0           STA   COUNTR       ; do it four times
C031  A2  00               LDX   [illegible]  ; start at CHCOPY+0
C033  A9  04       OUTLOP  LDA   #4           ; four ROLs
C035  8D  94  C0           STA   COUNT2       ; need a separate counter
C038  A9  00       INLOOP  LDA   #0
C03A  1E  8B  C0           ASL   CHCOPY,X     ; get carry
C03D  2A                   ROL                ; put in .A
C03E  1E  8B  C0           ASL   CHCOPY,X     ; again
C041  2A                   ROL                ; push it over
C042  E8                   INX                ; go up to next byte
C043  1E  8B  C0           ASL   CHCOPY,X
C046  2A                   ROL                ; more into .A
C047  1E  8B  C0           ASL   CHCOPY,X
C04A  2A                   ROL                ; now we have a number 0-15
                                              ;
C04B  A8                   TAY                ; put it in .Y for lookup
C04C  B9  6B  C0           LDA   OFFON,Y
C04F  20  D2  FF           JSR   CHROUT       ; print RVS ON or RVS OFF
C052  B9  7B  C0           LDA   QSCHAR,Y
C055  20  D2  FF           JSR   CHROUT       ; print the character
C058  CA                   DEX                ; back to normal
C059  CE  94  C0           DEC   COUNT2
C05C  D0  DA               BNE   INLOOP       ; continue for four characters
C05E  A9  0D               LDA   #13          ; return
C060  20  D2  FF           JSR   CHROUT       ; new line
C063  E8                   INX
C064  E8                   INX                ; increment .X by 2
C065  CE  93  C0           DEC   COUNTR       ; decrement outer loop
C068  D0  C9               BNE   OUTLOP       ; and go back again
C06A  60                   RTS
                                              ;
C06B  92  92  92   OFFON   .BYTE 146,146,146,146,146,18,18,18
C073  92  92  92           .BYTE 146,146,146,18,18,18,18,18
C07B  20  AC  BB   QSCHAR  .BYTE 32,172,187,162,188,161,191,190
C083  BE  BF  A1           .BYTE 190,191,161,188,162,187,172,32
C08B               CHCOPY  =     *
C093                       *=    * + 8
C093               COUNTR  =     *
C094                       *=    * + 1
C094               COUNT2  =     *
C095                       *=    * + 1
```

See also CHARX8.

Name

Print large (8 × 8) characters

Description

CHARX8 prints a gigantic character, eight times larger than normal. It's not especially useful as a screen routine (except perhaps for a children's alphabet program), but if you send output to a printer, you can use it to print large banners.

Prototype

1. Enter this routine with the screen code in .A and the carry flag clear to print a character from the uppercase/graphics character set, or with the carry flag set for a character from the uppercase/lowercase character set.
2. Store the screen code in zero page.
3. Manipulate a zero-page pointer to point to the character shape in ROM.
4. Switch in character ROM and copy the eight bytes to normal memory.
5. Loop through the eight bits of each of the eight bytes.
6. Print a reversed space for bits that are on, and a space for off bits.

Explanation

Patterns for the uppercase/graphics character set are stored in character ROM at $D000–$D7FF, while patterns for the uppercase/lowercase character set are found at $D800–$DFFF. Each of the 256 printable character patterns takes up eight bytes of memory, so a screen code value must be multiplied by 8 and then added to either $D000 or $D800 to calculate the starting address of the corresponding character pattern data. Once you have the memory address of the character shape, you can convert it into a big character.

FREEZP at location $FB is a pointer to the character shape we want to print. The accumulator holds the screen code, so first we have to store it in the low byte of FREEZP—to be multiplied by 8 in a moment. Next, the high byte of the pointer is set up. At $C002, the number $0D is loaded and stored into FREEZP+1. Next, the contents of the carry flag are rotated into the same location. At this point, both FREEZP and FREEZP+1 are three left-shifts away from pointing to the right place. A left-shift is the same as multiplying by 2, so three shifts are the same as "times 2 times 2 times 2," or "times 8."

ASLing the low byte followed by ROLing the high byte multiplies a number by 2, so we do that three times. The result is a two-byte pointer that tells us where to find the character.

At $C014–$C02B, we read the character shape. Memory at $D000–$DFFF is very busy: Character ROM is there, I/O locations are there, and RAM is there, too. Location 1 controls what's going on, and we have to turn off bit 2 to get to the character shapes. But, first, SEI turns off interrupts, so there's no need to worry about crashes. A loop copies the characters from ROM down to a section of memory we've set aside. CLI turns on the interrupts again.

Now we have the shape at CHCOPY within the program. There are eight bytes there, each of which contains eight bits. All that's left is to ROL the appropriate byte. The current high bit moves into the carry flag, and BCS branches to the print routine that prints a reversed space (if that's what is needed). Otherwise, the bit is cleared, and we need to print a normal space. After eight rotates, a CHR$(13) puts the cursor on the next line, and the outer loop continues until the last bit is converted into a space or reversed space.

Note: On the 64, it's necessary to turn off bit 2 of location 1 to get to the character set in ROM. On the 128, you can access the character by switching to bank 14. Thus, it's necessary to remove the instructions from SEI to STA $01 ($C014–$C01A) and the instructions from LDA $01 to CLI ($C025–$C02B). Also, the instruction LDA (FREEZP),Y at $C01D should be replaced with a call to the INDFET (indirect fetch) Kernal routine, as follows:

```
LOOP LDA #FREEZP  ; the zero page pointer
     LDX #14      ; the bank to access
     JSR  65396   ; the INDFET Kernal routine
```

Routine

```
C000             FREEZP  =    $FB
C000             CHROUT  =    $FFD2
                                          ; Enter with screen code in A.
                                          ; Carry clear for uppercase/graphics, carry set
                                          ; for uppercase/lowercase
                                          ;
C000  85  FB     CHARX8  STA  FREEZP      ; the screen code (low-byte, to multiply by 8)
C002  A9  0D             LDA  #%00001101  ; $0D, which will be shifted four times, to
                                          ; become $D0 or $D8
C004  85  FC             STA  FREEZP+1    ; almost ready to rotate
C006  26  FC             ROL  FREEZP+1    ; got carry now
                                          ; Now multiply it by 8.
                                          ;
```

```
C008  06  FB              ASL   FREEZP
C00A  26  FC              ROL   FREEZP+1
C00C  06  FB              ASL   FREEZP
C00E  26  FC              ROL   FREEZP+1
C010  06  FB              ASL   FREEZP
C012  26  FC              ROL   FREEZP+1
                                            ; FREEZP now points to the first byte of
                                            ; character pattern in ROM.
                                            ;
C014  78                  SEI               ; turn off interrupts while we read character
                                            ; ROM
C015  A5  01              LDA   $01         ; location 1, bit 2 controls character ROM
C017  29  FB              AND   #%11111011  ; mask it out to make the characters visible
C019  85  01              STA   $01
C01B  A0  07              LDY   #7          ; need eight bytes (0-7)
C01D  B1  FB      LOOP    LDA   (FREEZP),Y  ; get the shape
C01F  99  65  C0          STA   CHCOPY,Y    ; and put it in memory
C022  88                  DEY               ; count down
C023  10  F8              BPL   LOOP        ; we want #0, so count down to $FF
C025  A5  01              LDA   $01         ; check location 1
C027  09  04              ORA   #%00000100  ; and turn the bit back on
C029  85  01              STA   $01
C02B  58                  CLI               ; interrupts are OK now
                                            ;
                                            ; Now print the shape on the screen.
C02C  A9  0D              LDA   #13         ; carriage return
C02E  20  D2  FF          JSR   CHROUT      ; so we start on a new line
C031  A2  FF              LDX   #255        ; X must be the counter, because ROL doesn't
                                            ; accept Y-index
C033  E8          OUTLOP  INX               ; increment up to zero the first time
C034  E0  [illegible]     CPX   #8          ; after 0-7, we're done
C036  D0  01              BNE   START       ; not 8, so do a row
C038  60                  RTS               ; else we're done
C039  A0  [illegible]  START  LDY   #9      ; counter (eight loops)
C03B  [illegible]  INLOOP DEY               ; counting down to zero
C03C  D0  08              BNE   DOLINE
C03E  A9  0D              LDA   #13         ; print RETURN
C040  [illegible]  D2  FF JSR   CHROUT
C043  4C  33  C0          JMP   OUTLOP
C046  3E  65  C0  DOLINE  ROL   CHCOPY,X    ; move the top bit into carry
C049  [illegible] [illegible]  BCS   REVERS ; if it's a 1, carry is set
[illegible]  A9  92       LDA   #146        ; reverse off
C04D  20  D2  FF          JSR   CHROUT      ; print it
C050  A9  20              LDA   #32         ; character code for space
C052  [illegible]  D2  FF JSR   CHROUT      ; print it, too
C055  4C  [illegible]  C0 JMP   INLOOP
C058  A9  12      REVERS  LDA   #18         ; reverse on
C05A  20  D2  FF          JSR   CHROUT      ; print it
C05D  A9  20              LDA   #32         ; character code for space
C05F  20  D2  FF          JSR   CHROUT      ; print it
C062  4C  3B  C0          JMP   INLOOP
C065              CHCOPY  =     *
C06D                      *=    *+8         ; reserve eight bytes for a copy of the character
```

See also CHARX4.

Name

Check peripheral status via location 144

Description

The ML equivalent of BASIC's ST (status) variable is location 144 ($90). In general, if the value in location 144 isn't zero, something has gone wrong (usually, *end of file* or *device not present*).

Prototype

1. Load the accumulator from location 144.
2. Branch if equal to zero (BEQ) to continue the routine.
3. If not equal to zero, something has gone wrong.

Explanation

The following program attempts to open a file that doesn't exist. The BEQ should not occur. The letter *A* is printed, which means something has gone wrong.

Routine

```
C000                STAT     =    144         ; location 144 holds the status byte
C000                SETLFS   =    $FFBA
C000                SETNAM   =    $FFBD
C000                OPEN     =    $FFC0
C000                CHROUT   =    $FFD2
C000                CHKOUT   =    $FFC9
C000                CLRCHN   =    $FFCC
C000                CLOSE    =    $FFC3

C000  A9 02                  LDA  #2
C002  A2 08                  LDX  #8
C004  A0 02                  LDY  #2
C006  20 BA FF               JSR  SETLFS      ; set logical file 2,8,2
C009  A9 00                  LDA  #0
C00B  20 BD FF               JSR  SETNAM      ; no name
C00E  20 C0 FF               JSR  OPEN        ; open it
C011  A2 02                  LDX  #2
C013  20 C9 FF               JSR  CHKOUT      ; get ready to print
C016  A5 ■          CHK144   LDA  STAT        ; check the status
C018  F0 ■                   BEQ  FINIS       ; if equal to zero, OK
C01A  20 CC FF               JSR  CLRCHN      ; clear channels before printing
C01D  A9 41                  LDA  #65
C01F  20 D2 FF               JSR  CHROUT      ; print a letter A
C022  20 CC FF      FINIS    JSR  CLRCHN      ; clear all channels
C025  A9 02                  LDA  #2
C027  20 C3 FF               JSR  CLOSE       ; and close file 2
C02A  60                     RTS
```

See also DERRCK, RDSTAT.

CHOUTP

Name

Change the target screen memory address for CHROUT

Description

If you've relocated your text screen, any characters you print with CHROUT will be placed in the normal screen memory area unless you update the text screen pointer HIBASE. **CHOUTP** changes the pointer so that CHROUT (or PRINT in BASIC) print characters on the relocated screen.

Prototype

1. Enter this routine with .A containing the 1K text-screen offset (2 for 2K offset, and so forth).
2. Multiply .A by 4 to put HIBASE on an even 1K boundary.
3. Store the result in HIBASE.

Explanation

In the example, the text-screen pointer is changed to 8192. Using CHROUT, 500 bytes beginning at this location are filled with zeros. Printing CHR$(64)—the @ symbol—causes zeros to be POKEd into these locations (the screen code for @ is 0).

In the routine, SCRPTR represents the actual location (times 1K) of the text screen (that is, SCRPTR .BYTE 8 signifies that the screen begins at 8K, or location 8192).

On the 128, we home the cursor twice within the main program. This closes any text windows that may be opened and places the cursor at the top of the screen.

Routine

```
C000                HIBASE   =    648       ; HIBASE = 2619 on the 128—starting page
                                            ; of screen memory
C000                CHROUT   =    65490     ; Kernal character output routine
                                            ;
                                            ; Using CHROUT, fill 500 bytes beginning at
                                            ; 8192 with zeros.
C000  AD 27 C0               LDA  SCRPTR    ; .A contains 1K times SCRPTR offset
C003  20 21 C0               JSR  CHOUTP    ; change the PRINT location
C006  A9 13                  LDA  #19       ; HOME the cursor
C008  20 D2 FF               JSR  CHROUT
                                            ; JSR CHROUT, (128 only—to close any
                                            ; windows)
                                            ; Fill 500 bytes at the start of the new screen
                                            ; with zeros.
C00B  AD 28 C0               LDA  CHAR
C00E  A2 02                  LDX  #2
C010  A0 FA         OUTLOP   LDY  #250
C012  20 D2 FF      INLOOP   JSR  CHROUT
C015  88                     DEY
C016  D0 FA                  BNE  INLOOP    ; fill 250 bytes
C018  CA                     DEX
C019  D0 F5                  BNE  OUTLOP    ; do OUTLOP twice—2 times 250
C01B  A9 01                  LDA  #1        ; return to default screen at 1K
```

```
C01D  20 21 C0            JSR   CHOUTP
C020  60                  RTS
                                          ;
                                          ; Change screen base address for PRINT.
                                          ;  A holds 1K offset.
C021  0A         CHOUTP   ASL             ; multiply .A by 4 so HIBASE times 256
                                          ; puts us on a 1K boundary
C022  0A                  ASL
C023  8D 88 02            STA   HIBASE    ; now change the PRINT location
C026  60                  RTS
                                          ;
C027  08         SCRPTR   .BYTE 8         ; to print on a screen at 8K (8192)
C028  40         CHAR     .BYTE 64        ; character to print—here @
```

See also VIDBNK, VICADR.

Name

Character redefinition

Description

CHRDEF moves either one or both ROM character sets into RAM and redefines a series of characters within one of these sets.

Prototype

1. Before assembling this routine, list the screen codes of the characters you wish to redefine as SCCODE and provide the number of these characters as NUMDEF. Store the 1K offset for your RAM character set in CHROFF. Then list character data at the end of the routine beginning at CHRDAT. Define PAGCTR as 16 if you want to copy both ROM character sets. In this case, add the commented line, ADC #8, just after ADC ZT at $C066 if the characters you're redefining are in the second character set. On the 128, define VMCSB as 2604 rather than 53272.
2. Temporarily store the high byte of the offset address for the RAM character set in zero page (ZT).
3. Save the high byte of the ROM character set address in ZP+1.
4. Multiply the current video bank (0–3) by 64 to get the high byte of its starting address and add the high byte in ZT to this.
5. Store the result representing the high byte for the starting location of the RAM character set in ZP+3 and also in ZT for use in character redefinition.
6. Store a zero in the low byte, zero-page pointers to the ROM and RAM character sets.
7. Copy the ROM set from the address in ZP to the address in ZP+2. On the 64, set interrupts and switch in character ROM at 53248 before doing this. On the 128, copy the ROM set from memory bank 14 with INDFET.
8. When the copying process is complete, on the 64, switch back in the I/O at 53248 and clear interrupts. On both computers, point the VIC chip memory control register (or its shadow, on the 128) to the RAM character set.
9. To locate the characters being redefined in the RAM character set, multiply the screen code for each by 8 and add the result to the starting location for the set (in ZT).
10. Load eight bytes of data representing each redefined

character, and store this data beginning at the address determined in Step 9.
11. Repeat Steps 9 and 10 for all characters being redefined, and then RTS.

Explanation

In the program below, **CHRDEF** copies the uppercase/graphics character set—2K of character data—from ROM beginning at 53248 to RAM beginning at 14336 (assuming the current video bank is 0), and then redefines the left arrow (←) character to 1/8 and the ampersand (&) to 1/4. To copy the lowercase/uppercase set instead, replace LDA #>UPPGRP at $C006 with LDA #<LOWUPP.

To move both character sets from ROM, you need to allow room in the current 16K video bank for 4K of character data. To do this in the example program below, before assembling the program, change CHROFF in the equates to 12; this offsets the RAM character sets by 12288 in the current video bank. Also change PAGCTR to 16 to move 12*256, or 4096, bytes and, if the characters you're redefining belong in the second set, insert the commented instruction, ADC #8, near the end of the program. This instruction adds an additional 8K to the offset for the RAM character set and causes data for the redefined characters to be stored into the second set.

As it's currently set up, the program redefines just two characters—the left arrow (character 31) and the ampersand (character 38)—in the primary character set. But with **CHRDEF**, you can redefine up to 256 characters within *one* character set. Just define NUMDEF to the number of characters you want to redefine and list their screen codes at SCCODE. Then provide the eight bytes of pixel data for each character at CHRDEF.

By listing the character definition data in binary form, you can see how the new characters will appear on the screen. For instance, look at the data in $C08C–$C093, and you'll see the image of 1/8 used to redefine the left arrow.

Routine

```
C000        VMCSB    =   53272    ; VMCSB = 2604 on the 128—VIC II chip
                                  ; memory control register
C000        CIACRA   =   56334    ; interrupt control register A
C000        CI2PRA   =   56576    ; CIA #2 data port register A
C000        ZT       =   163      ; temporary zero-page storage (normally for
                                  ; tape and serial I/O)
C000        ZP       =   251
```

```
C000                UPPGRP  =    53248        ; address of uppercase/graphics ROM
                                              ; character set
C000                LOWUPP  =    55296        ; address of lowercase/uppercase ROM
                                              ; character set
C000                CHROFF  =    %00001110    ; 1K RAM character set offset in current video
                                              ; bank
C000                INDFET  =    65396        ; Kernal routine to fetch bytes indirectly from
                                              ; another bank (128 only)
                                              ;
                                              ; Put character set in RAM at 14K, redefine
                                              ; the ← and & characters.
                                              ; First move character set to RAM.
C000  A9 0E         CHRDEF  LDA  #CHROFF      ; load character set offset
C002  0A                    ASL               ; multiply by 4 to get high byte of
                                              ; character set offset
C003  0A                    ASL
C004  85 A3                 STA  ZT           ; store temporarily
C006  A9 D0                 LDA  #>UPPGRP     ; change UPPGRP to LOWUPP to move
                                              ; lowercase/uppercase ROM set
C008  85 FC                 STA  ZP+1         ; save high byte address of ROM character
                                              ; set
C00A  AD 00 DD              LDA  CI2PRA       ; get current 16K video bank
C00D  29 03                 AND  #%00000011   ; bank number is in bits 0-1
C00F  49 03                 EOR  #%00000011   ; to get actual bank number, 0-3
C011  0A                    ASL               ; multiply bank number by 64 to get the
                                              ; high byte of bank address
C012  0A                    ASL
C013  0A                    ASL
C014  0A                    ASL
C015  0A                    ASL
C016  0A                    ASL
C017  05 A3                 ORA  ZT           ; now, add the high byte of RAM character
                                              ; set offset to this
C019  85 FE                 STA  ZP+3         ; store the result (high byte address of
                                              ; RAM character set)
C01B  85 A3                 STA  ZT           ; save it for redefining characters below
C01D  A9 00                 LDA  #0           ; ROM and RAM set addresses are on
                                              ; even-page boundaries
C01F  85 FB                 STA  ZP           ; store 0 into low-byte address of ROM set
C021  85 FD                 STA  ZP+2         ; also into low-byte address of RAM set
                                              ; Now copy character set from ROM to
                                              ; RAM.
C023  78                    SEI               ; disable IRQ interrupts (64 only)
C024  A5 01                 LDA  1            ; select character ROM using configuration
                                              ; register (64 only)
C026  29 FB                 AND  #%11111011   ; clear bit 2 (64 only)
C028  85 01                 STA  1            ; reset configuration register (64 only)
                                              ; Now move character set(s) from ROM to
                                              ; RAM.
C02A  A0 00                 LDY  #0           ; initialize .Y as index
C02C  B1 FB         CMLOOP  LDA  (ZP),Y       ; from ROM location (64 only)
                                              ; Substitute next three lines for previous
                                              ; line on the 128.
                                              ; CMLOOP LDA #ZP
                                              ; LDX #14; bank number
                                              ; JSR INDFET; fetch character data from
                                              ; bank 14
                                              ;
C02E  91 FD                 STA  (ZP+2),Y     ; to RAM location
C030  C8                    INY               ; next byte
C031  D0 F9                 BNE  CMLOOP       ; move another 256 bytes
C033  E6 FC                 INC  ZP+1         ; next 256-byte block
C035  E6 FE                 INC  ZP+3
C037  CE 87 C0              DEC  PAGCTR       ; next page
```

```
C03A  D0 F0              BNE   CMLOOP       ; move all 256-byte blocks
C03C  A5 01              LDA   1            ; (64 only)
C03E  09 04              ORA   #%00000100   ; set configuration register to enable I/O
                                            ; (64 only)
C040  85 01              STA   1            ; reset register (64 only)
C042  58                 CLI                ; reenable interrupts (64 only)
C043  AD 18 D0           LDA   VMCSB        ; now, point VIC chip to RAM character set
C046  29 F0              AND   #%11110000   ; retain current 4-7 bits of VMCSB (text
                                            ; offset)
C048  09 0E              ORA   #CHROFF      ; or in bits 0-3 representing RAM character
                                            ; set offset
C04A  8D 18 D0           STA   VMCSB        ; and store result in control register
                                            ;
                                            ; Now redefine RAM characters.
                                            ; First calculate location of each character
                                            ; in RAM set.
C04D  A2 00              LDX   #0           ; let .X count number of characters that
                                            ; have been redefined
C04F  BD 8A C0  RDFLOP   LDA   SCCODE,X     ; load each character number to redefine
C052  85 FB              STA   ZP
C054  A9 00              LDA   [illegible]  ; clear high byte for ROL
C056  85 FC              STA   ZP+1
C058  06 FB              ASL   ZP           ; multiply SCCODE by 8 since eight bytes
                                            ; per character
C05A  26 FC              ROL   ZP+1
C05C  06 FB              ASL   ZP
C05E  26 FC              ROL   ZP+1
C060  06 FB              ASL   ZP
C062  26 FC              ROL   ZP+1
C064  A5 FC              LDA   ZP+1         ; now add start of RAM character set (carry
                                            ; cleared by last ROL)
C066  65 A3              ADC   ZT           ; only add high byte since character set is
                                            ; on a page boundary
                                            ; ADC #8; add 2K if you transfer both sets
                                            ; and characters are in second set
C068  85 FC              STA   ZP+1         ; specific character's address is now at ZP
C06A  8A                 TXA                ; store .X on stack temporarily
C06B  48                 PHA
C06C  A0 00              LDY   #0           ; index rows of pixels in one character
C06E  AE 88 C0  CHLOOP   LDX   ROWCTR       ; .X now contains pixel row
C071  BD 8C C0           LDA   CHRDAT,X     ; get next row of character data
C074  91 FB              STA   (ZP),Y       ; store into RAM set
C076  EE 88 C0           INC   ROWCTR       ; next row of data
C079  C8                 INY                ; next row for this character
C07A  C0 08              CPY   #8           ; do eight rows of this character
C07C  90 F0              BCC   CHLOOP       ; all done
C07E  68                 PLA                ; restore .X to contain number of characters
                                            ; that have been redefined
C07F  AA                 TAX
C080  [illegible]        INX                ; next character
C081  EC 89 C0           CPX   NUMDEF       ; have all characters been done?
C084  D0 C9              BNE   RDFLOP       ; if not, do another one
C086  60                 RTS                ; we're finished
                                            ;
C087  08        PAGCTR   .BYTE 8            ; move 8*256=2048 bytes (1 set); use 16 to
                                            ; move both sets
C088  00        ROWCTR   .BYTE 0            ; counter for row of pixel data
C089  02        NUMDEF   .BYTE 2            ; number of characters to redefine
C08A  1F 26     SCCODE   .BYTE 31,38        ; screen codes of character to redefine
C08C            CHRDAT   =     *            ; pixel data for ← (1/8)
C08C  40                 .BYTE %01000000
C08D  44                 .BYTE %01000100
C08E  48                 .BYTE %01001000
```

```
C08F  12                      .BYTE  %00010010
C090  25                      .BYTE  %00100101
C091  42                      .BYTE  %01000010
C092  05                      .BYTE  %00000101
C093  02                      .BYTE  %00000010
                                                  ; pixel data for & (1/4)
C094  40                      .BYTE  %01000000
C095  44                      .BYTE  %01000100
C096  48                      .BYTE  %01001000
C097  12                      .BYTE  %00010010
C098  26                      .BYTE  %00100110
C099  4A                      .BYTE  %01001010
C09A  1F                      .BYTE  %00011111
C09B  02                      .BYTE  %00000010
```

See also ANIMAT, CUST80.

Name

Get a character within a range

Description

CHRGTR will come in handy anytime you wish to limit the user's response to a specified range of characters. For instance, suppose you ask the user a question that requires a numeric response. Or suppose you want only alphabetic input.

In either case, this routine is ideal. You simply set the upper and lower limits of acceptable ASCII characters beforehand and JSR to **CHRGTR**.

Prototype

1. Set up the lower and upper (plus one) values of the ASCII character range (RANGE1 and RANGE2, respectively).
2. Get a keypress.
3. Compare its ASCII value to the lower delimiter (RANGE1).
4. If it's less, branch to step 2.
5. Compare its ASCII value to the upper delimiter (RANGE2).
6. If it's greater, branch to step 2.
7. Otherwise, return the acceptable ASCII character in .A.

Explanation

The example program is set up so that only letters between *A* and *Z* are accepted. To limit the input to number keys, change RANGE1 to 48 (ASCII 0) and RANGE2 to 58 (ASCII 9, plus 1).

Routine

```
C000                GETIN   =    65508
C000                CHROUT  =    65490
                                              ; Accept only keys in the range A–Z and
                                              ; print the keypress.
C000  20 07 C0              JSR  CHRGTR       ; get a character within a range
C003  20 D2 FF              JSR  CHROUT       ; print the character
C006  60                    RTS
                                              ;
                                              ; Get a character from within
                                              ; RANGE1 RANGE2
                                              ; return it in .A.
C007  20 E4 FF      CHRGTR  JSR  GETIN        ; get ASCII key
C00A  CD 15 C0              CMP  RANGE1       ; compare with RANGE1
C00D  90 F8                 BCC  CHRGTR       ; too low, so get another keypress
C00F  CD 16 C0              CMP  RANGE2       ; compare with RANGE2 plus 1
C012  B0 F3                 BCS  CHRGTR       ; too high, so get another key
C014  60                    RTS
                                              ;
C015  41            RANGE1  .BYTE 65          ; ASCII A
C016  5B            RANGE2  .BYTE 91          ; ASCII Z plus 1
```

See also BUFCLR, CHRGTS, CHRKER, MATGET.

Name

Get a specific character

Description

There will be many occasions when you will want to screen the user's input selectively. Probably the most common example of this is when you ask the user a yes/no question. Usually, all you're really looking for is a Y or N response.

By using **CHRGTS**, you can set up this situation with ease. Before you access the routine, just place these two characters in the table of acceptable responses at the end of the program.

CHRGTS checks the incoming character to insure that it is among those in your table of allowed characters. The program continues only if and when it receives a suitable response.

Prototype

1. Get a keypress.
2. Compare its ASCII value with a list of acceptable responses (here, KEYS).
3. If the incoming keypress is among those in the table, return its ASCII value in .A.
4. Otherwise, branch to step 1.

Explanation

With the aid of **CHRGTS**, the following program checks for a Y (yes) or N (no) keypress. If either is pressed, it is printed. Otherwise, the program fetches another keypress until a Y or N is received.

Note: The table of acceptable responses can have as many ASCII characters in it as you like. By placing the responses that you're more likely to receive at the beginning of the table, you can speed up the execution of this routine.

Routine

```
C000                GETIN    =    65508
C000                CHROUT   =    65490
                                            ;
                                            ; Accept either Y or N only
C000  20  07  C0             JSR  CHRGTS    ; get specific characters
C003  20  D2  FF             JSR  CHROUT    ; print it
C006  60                     RTS
                                            ;
                                            ; Get only characters designated in KEYS
                                            ; table. Return character in .A.
```

```
C007  20 E4 FF  CHRGTS   JSR   GETIN       ; get ASCII key
C00A  A2 00               LDX   #0
C00C  DD 19 C0  CHKLOP   CMP   KEYS,X      ; check each character in table
C00F  F0 07               BEQ   EXIT        ; if found
C011  [redacted]          INX
C012  E0 02               CPX   #NUMKEY     ; check key number
C014  D0 F6               BNE   CHKLOP      ; if more in table, check next character
C016  F0 EF               BEQ   CHRGTS      ; if no match, get another keypress
C018  60        EXIT     RTS
                                            ;
C019  59 4E     KEYS      ASC   "YN"        ; list of acceptable keystrokes
C01B            NUMKEY    =     * - KEYS    ; number of acceptable keys
```

See also BUFCLR, CHRGTR, CHRKER, MATGET.

Name

Get a character

Description

You'll find a need for this routine in just about any program you write that requires user input. **CHRKER** uses the Kernal routine GETIN to get a character from the current input device.

Prototype

1. JSR to GETIN to fetch a keypress.
2. If the Z flag is set—if GETIN has received a null string, or CHR$(0)—BEQ to step 1.
3. Otherwise, return in .A the ASCII character received by GETIN.

Explanation

The example program gets a character from the keyboard (by default, the current input device) and prints it.

Note: GETIN relies on the normal IRQ interrupt routine to get its characters. During each IRQ interrupt, the keyboard is checked, and ASCII values for keypresses are placed in the keyboard buffer. So, altering the normal IRQ routines may cause the keyboard buffer not to be updated. In such instances, GETIN won't work, and you should use the Kernal SCNKEY routine instead.

A CMP #0 instruction following JSR GETIN may be necessary when you're getting characters from a device other than the keyboard (for example, from a disk or modem).

Routine

```
C000                GETIN   =     65508      ; Kernal get-key routine
C000                CHROUT  =     65490
                                             ;
                                             ; Accept keypresses until RETURN
C000  20 0B C0 LOOP         JSR   CHRKER     ; get a key in .A
C003  20 D2 FF              JSR   CHROUT     ; print it
C006  C9 0D                 CMP   #13        ; is it RETURN?
C008  D0 F6                 BNE   LOOP       ; if not, get another keypress
C00A  60                    RTS
                                             ;
                                             ; Return a keypress in .A.
C00B  [redacted] E4 FF CHRKER JSR GETIN      ; get an ASCII keystroke
C00E  F0 [redacted]         BEQ   CHRKER     ; If no keypress, then loop
C010  [redacted]            RTS
```

See also BUFCLR, CHRGTR, CHRGTS, MATGET.

Name

Convert signed integers to floating point and vice versa

Description

A signed integer value consists of 16 bits (two bytes). The highest bit indicates the sign (%0 is positive, %1 is negative); the remaining 15 bits contain the value. Floating-point numbers may contain fractional components and are contained within five bytes. This routine converts between the two formats.

Prototype

1. JMP indirectly through $0005 (64) or $117C (128) to convert integers to floating point. Enter with the integer value in .A (low byte) and .Y (high byte). The resulting floating-point value will be left in FAC1 (floating point accumulator #1), locations $61–$65 (64) or $63–$67 (128).
2. Or JMP indirectly through $0003 (64) or $117A (128) to change floating-point numbers to integers. Enter with the floating-point value in FAC1 (floating point accumulator #1), locations $61–$65 (64) or $63–$67 (128). The integer value will be returned in .A (low byte) and .Y (high byte).

Explanation

The example program takes the two-byte value in the start of BASIC pointer, converts it to a floating-point number, calls the square-root routine, and prints the result. There's no good reason why you'd want to find the square root of the start of BASIC, of course, but it serves as a good example of using built-in ROM routines.

The RAM vectors to the built-in conversion routines in BASIC ROM are initialized when the computer is turned on or reset. The example also uses the ROM routine for the SQR function, which calculates the square root of the floating-point value in FAC1, and the ROM routine that prints a signed integer number.

A note to machine language programmers who want to use fractions and floating-point routines in their programs: There are a variety of ways to avoid fractions or to simulate them without going to floating point. If you're convinced that you need fractions, you may take one of two routes. The first is to use the various ROM routines; the second is to write your own floating-point package. If you depend on the BASIC routines, your programs will perform calculations at about the

same speed as a BASIC program, which is a good argument for using BASIC in the first place. Writing your own floating-point package is feasible, but it's a lot of work, and the end result may be a set of routines that aren't much faster than BASIC.

Note: 128 programmers should substitute the following addresses: SQR = $8FB7, LINPRT = $8E32, **CI2FP** = JMP ($117C), **CFP2I** = JMP ($117A).

Routine

```
C000              TXTTAB   =    43          ; TXTTAB = 45 on the 128—pointer to start
                                            ; of BASIC
C000              SQR      =    $BF71       ; ROM square-root routine (SQR = $8FB7 on
                                            ; the 128)
C000              LINPRT   =    $BDCD       ; LINPRT = $8E32 on the 128—prints
                                            ; signed integer in .A and .X
                                            ;
C000  A4 2B       MAIN     LDY  TXTTAB      ; low byte of the pointer
C002  A5 2C                LDA  TXTTAB+1    ; high byte
C004  20 15 C0             JSR  CI2FP       ; convert it
C007  20 71 BF             JSR  SQR         ; find the square root (ROM routine)
C00A  20 18 C0             JSR  CFP2I       ; back to an integer
C00D  48                   PHA              ; save .A
C00E  98                   TYA              ; .Y to .A
C00F  AA                   TAX              ; to .X
C010  68                   PLA              ; get .A back
C011  20 CD BD             JSR  LINPRT      ; print it
C014  60                   RTS
                                            ;
C015  6C 05 00    CI2FP    JMP  ($0005)     ; JMP ($117C) on the 128
                                            ;
C018  6C 03 00    CFP2I    JMP  ($0003)     ; JMP ($117A) on the 128
```

See also B2SNIN, B2UNIN, BCD2BY, CB2BCD, CI2FP, CNVBFP.

Name

Convert a two-byte integer to four hexadecimal (ASCII) digits

Description

This routine is just an extended two-byte version of **CB2HEX**, which converts a single byte into two hex characters. You enter **CI2HEX** with the high byte in .A, the low byte in .X. The result is stored in a buffer, terminated by a zero.

Prototype

1. With the high byte in .A and low byte in .X, call the byte-to-hex (BYTHEX) subroutine.
2. Copy the resulting characters (stored in zero page) to a buffer.
3. Transfer .X to .A and call BYTHEX again.
4. Copy the ASCII hex characters to the buffer again.

Explanation

The example routine displays a section of memory starting at $0800, where BASIC programs are stored on the 64. On the 128, programs are stored at $1C00 or $4000, depending on whether a graphics area has been allocated. To adapt the program to the 128, change the $08 at $C004 to $1C or $40.

The **CI2HEX** routine is called to set up the memory addresses ($0800, $0808, $0810, and so on) to be printed at the beginning of each line. Then eight single-byte values are printed, separated by spaces. The BYTHEX subroutine at $C07C is essentially the same as the **CB2HEX** routine found elsewhere in this book, but because the X and Y registers are used in the calling routines, BYTHEX is careful not to disturb any values in the registers.

The two ASCII characters are stored in $FD and $FE temporarily. The BUFFIT routine copies these characters to the buffer, indexed by .Y. Later, the PRBUFF routine prints out the characters in BUFFER.

Routine

```
C000             ZP       =    $FB
C000             F1       =    $FD
C000             F2       =    $FE
C000             CHROUT   =    $FFD2
C000  A9 00      MAIN     LDA  #0
C002  85 FB               STA  ZP
C004  A9 08               LDA  #8
C006  85 FC               STA  ZP+1      ; set up a pointer to $0800 in ZP
                                         ;
C008  A9 0A               LDA  #10       ; ten lines
C00A  8D 9D C0            STA  COUNTER   ; stash it in a memory variable
```

```
C00D  A6  FB        UTLOOP   LDX   ZP          ; low byte of pointer
C00F  A5  FC                 LDA   ZP + 1      ; high byte
C011  20  4B  C0             JSR   CI2HEX      ; convert it
C014  20  6E  C0             JSR   PRBUFF      ; print the buffer
C017  20  69  C0             JSR   PRSPC       ; print a space
C01A  A0  00                 LDY   #0
C01C  B1  FB        INLOOP   LDA   (ZP),Y
C01E  20  7C  C0             JSR   BYTHEX
C021  A5  FD                 LDA   F1
C023  20  D2  FF             JSR   CHROUT
C026  A5  FE                 LDA   F2
C028  20  D2  FF             JSR   CHROUT
C02B  20  69  C0             JSR   PRSPC
C02E  C8                     INY
C02F  C0  08                 CPY   #8
C031  D0  E9                 BNE   INLOOP
C033  A9  0D                 LDA   #13         ; print RETURN
C035  20  D2  FF             JSR   CHROUT
C038  A9  08                 LDA   #8          ; add 8 to the ZP pointer
C03A  18                     CLC               ; always CLC before adding
C03B  65  FB                 ADC   ZP          ; add
C03D  85  FB                 STA   ZP          ; store it back
C03F  A9  00                 LDA   #0
C041  65  FC                 ADC   ZP + 1      ; adding 0 takes care of carry
C043  85  FC                 STA   ZP + 1      ; store that, too
C045  CE  9D  C0             DEC   COUNTER     ; count down
C048  D0  C3                 BNE   UTLOOP      ; and branch back
C04A  60                     RTS               ; end of the main routine
                                               ;
C04B                CI2HEX   =     *
C04B  A0  00                 LDY   #0
C04D  20  7C  C0             JSR   BYTHEX      ; convert .A to hex in F1, F2
C050  20  57  C0             JSR   BUFFIT
C053  8A                     TXA
C054  20  7C  C0             JSR   BYTHEX
C057  A5  FD        BUFFIT   LDA   F1
C059  99  9E  C0             STA   BUFFER,Y
C05C  C8                     INY
C05D  A5  FE                 LDA   F2
C05F  99  9E  C0             STA   BUFFER,Y
C062  C8                     INY
C063  A9  00                 LDA   #0
C065  99  9E  C0             STA   BUFFER,Y
C068  60                     RTS
                                               ;
C069  A9  20        PRSPC    LDA   #32
C06B  4C  D2  FF             JMP   CHROUT      ; print a space
                                               ;
C06E  A0  00        PRBUFF   LDY   #0
C070  B9  9E  C0    PBLOOP   LDA   BUFFER,Y
C073  F0  06                 BEQ   OUT
C075  20  D2  FF             JSR   CHROUT
C078  C8                     INY
C079  D0  F5                 BNE   PBLOOP
C07B  60            OUT      RTS
C07C                BYTHEX   =     *
C07C  08                     PHP               ; save the processor status
C07D  48                     PHA               ; save .A
C07E  4A                     LSR
C07F  4A                     LSR
C080  4A                     LSR
C081  4A                     LSR               ; four shift rights, for the high nybble
C082  20  93  C0             JSR   ADD48       ; add 48 (plus 7, maybe)
C085  85  FD                 STA   F1          ; store it
```

```
C087  68                   PLA                ; pull .A for the low nybble
C088  48                   PHA                ; push one more time
C089  29 0F                AND   #%00001111   ; mask it
C08B  20 93 C0             JSR   ADD48        ; and add 48
C08E  85 FE                STA   F2           ; store it in F2
C090  68                   PLA                ; get .A back
C091  28                   PLP                ; and .P, too
C092  60                   RTS
                                              ;
C093  18          ADD48    CLC
C094  69 30                ADC   #48          ; add 48
C096  C9 3A                CMP   #58          ; Is it 0-9?
C098  90 02                BCC   NOMORE       ; yes, move ahead
C09A  69 06                ADC   #6           ; else, add 7 (with carry set)
C09C  60          NOMORE   RTS
                                              ;
C09D  00          COUNTER  .BYTE 0
C09E              BUFFER   =     *
C19D                       *=    *+255        ; a big buffer
```

See also BCD2AX, CAS2IN, CB2ASC, CB2HEX.

Name

Close a file and restore default devices

Description

This routine closes the logical file whose number is in the accumulator. It also restores the keyboard and screen as the current input and output devices.

CLOSFL can close any external channel (such as disk drive, printer, or modem) as long as the channel number is in .A.

Prototype

1. Load .A with the logical file number of the external device.
2. JSR to CLOSE.
3. JMP to CLRCHN.

Explanation

See **PRTOUT** or **PRTSTR** for programs where **CLOSFL** is used to close a printer channel. In the **WRITBF** and **READBF** routines, **CLOSFL** closes a channel to the disk after file writing or reading. No error will occur if you try to close a file which hasn't been opened.

Routine

```
C000                CLOSE    =     65475
C000                CLRCHN   =     65484

                                          ;
                                          ; CLOSFL closes the logical file in .A and
                                          ; restores default devices.
C000  20  C3  FF  CLOSFL   JSR   CLOSE    ; close file in .A
C003  4C  CC  FF           JMP   CLRCHN   ; clear all channels, restore default devices,
                                          ; and RTS
```

See also OPENPR, PRTOUT, PRTSTR, WRITBF, WRITFL.

Name

Clear the screen with CHR$(147)

Description

One of three routines in this book that clears the text screen, this one accomplishes the task by printing CHR$(147), the Commodore ASCII code for clearing the screen.

Prototype

Load .A with 147 and JMP to CHROUT.

Explanation

This simple program clears the text screen and prints a *Y* in the current cursor color.

Note: This routine is much faster than **CLRFIL**, but just slightly slower than **CLRROM**. Unlike **CLRROM**, though, it has the advantage of relying on a Kernal ROM routine, specifically CHROUT. And like other ROM routines accessed from the Kernal jump table, CHROUT will be called from the same address on all Commodore machines.

Routine

```
C000                 CHROUT  =    65490
                                              ;
                                              ; Clear screen and print Y.
C000  20  09  C0             JSR  CLRCHR      ; clear the screen
C003  A9  59                 LDA  #89         ; print Y
C005  20  D2  FF             JSR  CHROUT
C008  60                     RTS
                                              ;
                                              ; Clear the screen with CHR$(147).
C009  A9  93         CLRCHR  LDA  #147        ; print CLEAR SCREEN
C00B  4C  D2  FF             JMP  CHROUT      ; and RTS
```

See also CLRFIL, CLRROM.

Name

Clear the screen with a fill routine

Description

Yet another routine to clear the text screen, this one works by storing a 32 (the screen code for the space character) into each screen memory location.

Prototype

Using a loop, store spaces in all 1000 text-screen locations.

Explanation

This short program clears the text screen by filling it with spaces, then prints an X.

Note: This routine leaves color memory unchanged. If you wish to fill color memory at the same time the screen is cleared, insert a JSR **COLFIL** in the code following the fill loop and add **COLFIL** to the end of the program.

You may notice that the BNE occurs after the STAs in the primary loop, instead of in its more natural position just after a DEY. The STA instruction does not affect any flags; the BNE refers back to the DEY just after the LDY. The four store instructions must store in offsets of 0–249. By performing the STAs before the BNE, we're able to store in the offset of zero.

Routine

```
C000                SCREEN   =     1024          ; normal text-screen position
C000                CHROUT   =     65490
                                                 ;
                                                 ; Clear screen with fill and print X.
C000  20 09 C0               JSR   CLRFIL        ; clear the screen
C003  A9 58                  LDA   #88           ; print X
C005  20 D2 FF               JSR   CHROUT
C008  60                     RTS
                                                 ;
C009  A9 20         CLRFIL   LDA   #32           ; screen code for space
C00B  A0 FA                  LDY   #250
C00D  88            LOOP     DEY
C00E  99 00 04               STA   SCREEN,Y      ; 1st quarter
C011  99 FA 04               STA   SCREEN+250,Y  ; 2nd quarter
C014  99 F4 05               STA   SCREEN+500,Y  ; 3rd quarter
C017  99 EE 06               STA   SCREEN+750,Y  ; 4th quarter
C01A  D0 F1                  BNE   LOOP          ; fill all 250 bytes
                                                 ; Insert JSR COLFIL to fill color RAM as
                                                 ; well.
C01C  60                     RTS
```

See also CLRCHR, CLRROM.

Name

Clear the hi-res screen using a fill method

Description

Anytime you display the high-resolution screen without first clearing it, you're likely to see whatever garbage resides in the underlying memory. To avoid this, clear screen memory with the **CLRHRF** routine, or with **CLRHRS**, before you view it.

The routine shown here relies on a conventional zero-page addressing technique to fill 8192 bytes representing screen memory with zeros. **CLRHRS** achieves the same result, but in slightly less time and with less memory, by using self-modifying code.

With either method, high-resolution color memory remains intact. If you want to fill color memory at the same time, insert a JSR **HRCOLF** into your code where indicated.

Prototype

1. Store the address of the high-resolution screen in a zero-page pointer.
2. Set .X to 32 as a counter for the number of pages to fill (32 * 256 = 8192).
3. Using indirect indexed addressing, fill each byte within a page with zero (in .A).
4. After filling a page, increment the page pointer in zero page.
5. Decrement .X. If it's not equal to zero, go to step 3.
6. When .X = 0, RTS to the main program. (If you want to clear color memory as well, JSR to **HRCOLF** just before the RTS.)

Explanation

In the example program, we set up a high-resolution screen at location 8192 and clear it by using **CLRHRF**. A keypress returns you to the normal text screen.

On the 64, before locating the bitmap within the current video bank (by default, bank 0), you must save the contents of the VIC-II chip memory control register at 53272 (VMCSB). This register contains the present offset address within the current video bank for the character set (low nybble) and the text screen (high nybble).

On the 128, during each IRQ interrupt, VMCSB takes its value from either VM1 at 2604 (if you're in text mode) or from VM2 at 2605 (if you're in bitmap mode). Since VM1 is never

altered by the program, you don't need to save it (or VMCSB) here.

Next, bit 3 of VMCSB (VM2 on the 128) is turned on to offset the high-resolution screen by 8K within the current video bank. To place your screen in the first half of the video bank (the offset will be 0), turn off bit 3 by ANDing the contents of the control register with 247.

Once you've located the high-resolution screen, the subroutine **BITMAP** puts the screen in bitmap mode. The screen is then cleared with **CLRHRF**.

On the 64, returning to the normal text screen is actually a two-step procedure. After bitmap mode has been disabled (again with **BITMAP**), the contents of the VIC-II memory control register are restored so that they point to the character set and text screen that were previously in use. On the 128, because VMCSB takes its value from VM1 in text mode, you need only to disable bitmap mode.

Routine

```
C000                 ZP      =    251
C000                 GETIN   =    65508
C000                 VMCSB   =    53272        ; VIC II chip memory control register
C000                 SCROLY  =    53265        ; scroll/control register—use GRAPHM =
                                               ; 216 on the 128
C000                 VM2     =    2605         ; VIC II chip memory control shadow register
                                               ; (128 only)
                                               ;
                                               ; Locate a hi-res screen at 8192 and clear it
C000  AD 18 D0               LDA  VMCSB        ; temporarily save VMCSB (64 only)
C003  8D 45 C0               STA  TEMP         ; (64 only)
                                               ; Now, offset bitmap by 8K in video bank
C006  09 08                  ORA  #%00001000   ; replace with AND #%11110111 if hi-res
                                               ; screen is in first half of video bank
C008  8D 18 D0               STA  VMCSB        ; reset register (replace VMCSB with VM2 on
                                               ; the 128)
C00B  20 3A C0               JSR  BITMAP       ; enter bitmap mode
C00E  20 20 C0               JSR  CLRHRF       ; clear the hi-res screen
C011  20 E4 FF  WAIT         JSR  GETIN        ; get a keypress
C014  F0 FB                  BEQ  WAIT         ; if no keypress, wait
C016  20 3A C0               JSR  BITMAP       ; turn off bitmap mode
                                               ;
                                               ; Reset pointer to character set.
C019  AD 45 C0               LDA  TEMP         ; (64 only)
C01C  8D 18 D0               STA  VMCSB        ; (64 only)
C01F  60                     RTS
                                               ;
                                               ; Clear the hi-res screen with a fill method.
C020  AD 43 C0  CLRHRF       LDA  HRSCRN       ; set up zero-page pointers to the hi-res
                                               ; screen
C023  85 FB                  STA  ZP
C025  AD 44 C0               LDA  HRSCRN+1
C028  85 FC                  STA  ZP+1
                                               ; Fill 32 pages (8K) with zeros.
C02A  A9 00                  LDA  #0
C02C  A8                     TAY
```

```
C02D  A2  20              LDX   #32          ; 32 pages
C02F  91  FB      LOOP    STA   (ZP),Y       ; fill a block of 256 bytes with zero
C031  C8                  INY
C032  D0  [redacted]      BNE   LOOP
C034  E6  FC              INC   ZP+1         ; page filled, so increase page pointer
C036  CA                  DEX
C037  D0  F6              BNE   LOOP         ; to fill all pages
                                             ;
                                             ; JSR HRCOLF; Insert here to clear color
                                             ; memory as well.
                                             ;
C039  60                  RTS
                                             ;
                                             ; Enable/disable bitmap mode.
C03A  AD  11  D0  BITMAP  LDA   SCROLY       ; substitute GRAPHM for SCROLY for the
                                             ; 128
C03D  49  20              EOR   #%00100000   ; flip bit 5
C03F  8D  11  D0          STA   SCROLY       ; reset register (again, use GRAPHM instead
                                             ; of SCROLY for the 128)
C042  60                  RTS
                                             ;
C043  00  20      HRSCRN  .WORD 8192         ; locate hi-res screen
C045  00          TEMP    .BYTE 0            ; temporary storage for VMCSB configuration
```

See also BITMAP, CLRHRS, HRCOLF, HRPOLR, HRSETP, PAINT.

CLRHRS

Name

Clear a hi-res screen using self-modifying code

Description

This is probably the quickest way to clear the 8000 bytes of a hi-res screen.

Prototype

1. Store the address of the high-resolution screen in the dummy address (initially $FFFF) at $C012.
2. Set .X to 32, for the number of pages to fill (32 * 256 = 8192).
3. Fill each byte within a page with zero (in .A) using absolute addressing offset by .Y.
4. After filling a page, increment the high-byte page pointer in the absolute address.
5. Decrement .X. If it's not equal to zero, go to step 3.
6. When .X = 0, RTS to the main program. (If you also want to clear color memory, JSR to **HRCOLF** just prior to returning.)

Explanation

It might look confusing when you first read through the program, but the idea is reasonably simple. The line at $C011 is the key. It says STA $FFFF,Y, but that instruction never really happens. The first part of the program takes the address of the hi-res screen (8192, in this example) and stores it low byte first, just after the STA instruction.

The routine works by modifying itself, changing the address after the STA a total of 32 times.

Routine

```
C000  AD 1F  C0  CLRHRS   LDA   HRSCRN+1    ; store hi-res screen location in dummy
                                            ; location—$FFFF
C003  8D 13  C0           STA   LOOP+2
C006  AD 1E  C0           LDA   HRSCRN
C009  8D 12  C0           STA   LOOP+1
                                            ; Fill 32 pages (8K) with zeros.
C00C  A9 00               LDA   #0
C00E  A8                  TAY
C00F  A2 20               LDX   #32         ; 32 pages
C011  99 FF  FF  LOOP     STA   $FFFF,Y     ; fill a block of 256 bytes with zeros
C014  C8                  INY
C015  D0 FA               BNE   LOOP
```

```
C017  EE 13 C0           INC   LOOP+2     ; page filled, so increase high byte of
                                          ; pointer
C01A  CA                 DEX
C01B  D0 F4              BNE   LOOP       ; to fill all pages
                                          ; Insert JSR HRCOLF here to clear color
                                          ; memory as well.
C01D  60                 RTS
C01E  00 20     HRSCRN   .WORD 8192       ; hi-res screen
```

See also CLRFIL, CLRROM.

Name

Clear the screen with a ROM routine

Description

This is one of three routines in this book that is used for clearing the text screen. Each has advantages. This particular routine uses a Kernal ROM routine (labeled CLRHOM) located on the 64 at 58692. An equivalent routine is at 49474 on the 128.

Prototype

JMP to CLRHOM.

Explanation

This short program clears the text screen and prints a Z. The letter will print in the current cursor color.

Note: **CLRROM** is much faster than **CLRFIL** and slightly faster than **CLRCHR**. But, again, it relies on a ROM routine that may change locations on a later version of the 64 or 128.

Routine

```
C000                CLRHOM   =     58692      ; CLRHOM = 49474 on the 128
C000                CHROUT   =     65490
                                              ;
                                              ; Clear the screen and print Z.
C000  20  09  C0             JSR   CLRROM     ; clear the screen
C003  A9  5A                 LDA   #90        ; print Z
C005  20  D2  FF             JSR   CHROUT
C008  60                     RTS
                                              ;
                                              ; Clear the screen with a Kernal ROM
                                              ; routine,
C009  4C  44  E5    CLRROM   JMP   CLRHOM     ; and RTS
```

See also CLRCHR, CLRFIL.

Name

Print the value of a two-byte integer

Description

BASIC offers a built-in ROM routine for printing the value of a two-byte integer—LINPRT. We've shown how to use this routine in the discussion of **NUMOUT**, elsewhere in this book.

There will be times, however, when you'll find yourself working in a programming environment where it's inconvenient to access LINPRT—as when you're in RAM under BASIC ROM on the 64, or in a bank that doesn't contain BASIC on the 128. At other times, you may simply want to write a generic program that runs on both the 128 and the 64.

In either case, a custom routine like **CNUMOT** will give you this option.

Prototype

1. Prior to entering the routine, set up a table of two-byte subtrahends for each digit's place—1, 10, 100, 1000, and 10,000.
2. Enter this routine with the two-byte number to print in .X (low byte) and .A (high byte).
3. Save the low and high bytes of the integer in zero page locations.
4. Count the number of times the subtrahend representing the largest digit's place (10,000) can be subtracted from the value (in .X and .A) before a number less than zero results.
5. Print this number to the screen.
6. Repeat steps 4 and 5 for the remaining digit places—1000, 100, 10, and 1.

Explanation

With **CNUMOT**, we print the two-byte starting address of BASIC text.

Here, **CNUMOT** works much like our conversion routine for a one-byte integer (see **BYTASC**). Again, a subtraction method is used, only this time it handles a second byte as well. And instead of passing a single byte to the routine in .A as before, the low byte of the two-byte integer is sent to the routine in .X and the high byte in .A.

Although it takes some time to set up the routine, the basic idea is simple. First, subtract 10,000. Subtract it again and again until a negative number results. Now you know how many 10,000s fit into the number. Next, subtract 1000 as

many times as necessary. The third step is to subtract 100, then 10, then 1. At each stage, the program keeps track of how many times a given value has been subtracted and prints out the total.

In this case, the integer occupying a two-byte address must lie in a range from 0 through 65535. The number can have as many as five digits.

Begin with the highest digit for the number—here, the 10,000's place. We repeatedly subtract 10,000—the first entry in the table of two-byte subtrahends, or TB2SUB—from the two-byte number until a negative result occurs. For each subtraction that yields a positive value (>—0), increment the place-holder counter—kept here in the Y register.

When subtraction finally produces a negative value, the two-byte number itself is restored to the value it had before this last subtraction, and the ASCII equivalent of the digit in .Y printed within DONE.

This entire process is repeated for the next four digits (the 1000's place, the 100's place, the 10's place, and the 1's place).

A flag (ZEROFL) within the printing routine prevents leading zeros from being displayed. Only when this flag contains a nonzero value will the digit zero be printed. If ZEROFL is still zero after all five digits have been evaluated, we simply print a zero.

Note: There is one important difference between this routine and **BYTASC** when it comes to understanding the two. Here, each digit is printed after it has been converted, whereas with **BYTASC**, we wait to print the entire number after all digits have been converted.

Routine

```
C000                CHROUT  =    65490
C000                TXTTAB  =    43          ; TXTTAB = 45 on the 128—start-of-BASIC
                                             ; pointer
C000                ZP      =    251
                                             ;
                                             ; Print the start of BASIC
C000  A9 93         CLRCHR  LDA  #147        , clear the screen
C002  20 D2 FF              JSR  CHROUT
                                             ; Print the message
C005  A0 00                 LDY  #0          ; print "BASIC STARTS AT "
C007  B9 71 C0      LOOP    LDA  STRING,Y
C00A  F0 07                 BEQ  POINT       ; if zero byte, then don't print it
C00C  20 D2 FF              JSR  CHROUT
C00F  C8                    INY              ; next character
C010  4C 07 C0              JMP  LOOP        ; and continue
C013  A6 2B         POINT   LDX  TXTTAB      ; load low- and high-byte start-of-BASIC
                                             ; pointers
C015  A5 2C                 LDA  TXTTAB+1
```

```
C017  4C 1A C0            JMP   CNUMOT       ; convert two-byte integer to ASCII, print it,
                                             ; and RTS
                                             ;
                                             ; CNUMOT converts two-byte integer in
                                             ; .X (low) and .A (high byte) to ASCII and
                                             ; prints it
C01A  86 FB      CNUMOT   STX   ZP           ; save low and high byte of integer to zero
                                             ; page
C01C  85 FC               STA   ZP+1
C01E  A9 00               LDA   #0           ; initialize ZEROFL
C020  8D 82 C0            STA   ZEROFL
C023  A2 08               LDX   #8           ; index to TB2SUB table, initially points to
                                             ; low byte of 10000
C025  A0 FF      INITCT   LDY   #255         ; initialize counter for each digit's place
C027  C8         SUBTLP   INY                ; begin subtraction loop, counter starts with
                                             ; zero
C028  A5 FB               LDA   ZP
C02A  48                  PHA                ; save the low byte of number
C02B  38                  SEC
C02C  FD 67 C0            SBC   TB2SUB,X     ; subtract low byte of subtrahend from low
                                             ; byte of number
C02F  85 FB               STA   ZP           ; store result in zero page
C031  A5 FC               LDA   ZP+1         ; now do the same with high byte
C033  48                  PHA                ; save the high byte of the number
C034  FD 68 C0            SBC   TB2SUB+1,X   ; subtract high byte of subtrahend from
                                             ; high byte of number
C037  85 FC               STA   ZP+1         ; and store the result
C039  90 05               BCC   DONE         ; subtraction gave number less than zero,
                                             ; so we're done
C03B  68                  PLA                ; restore the stack
C03C  68                  PLA
C03D  4C 27 C0            JMP   SUBTLP       ; and continue subtraction
                                             ; Restore high and low bytes to values
                                             ; before we dropped below zero.
C040  68         DONE     PLA                ; pull high byte
C041  85 FC               STA   ZP+1         ; and store it
C043  68                  PLA                ; pull low byte of number
C044  85 FB               STA   ZP           ; and store it also
C046  98                  TYA                ; put digit's place counter into .A
C047  AC 82 C0            LDY   ZEROFL       ; determine whether a nonzero digit has
                                             ; occurred
C04A  D0 07               BNE   CNVERT       ; branch if a nonzero digit has been printed
C04C  C9 00               CMP   #0           ; check for zero
C04E  F0 08               BEQ   ZEROHI       ; don't print a zero if no nonzero digits
                                             ; have been printed
C050  8D 82 C0            STA   ZEROFL       ; change the flag to a nonzero value
C053  09 30      CNVERT   ORA   #48          ; convert digit's place counter to ASCII
C055  20 D2 FF            JSR   CHROUT       ; and print it
C058  CA         ZEROHI   DEX                ; decrement twice for each word in
                                             ; subtrahend table
C059  CA                  DEX
C05A  10 C9               BPL   INITCT       ; for the next place
C05C  AD 82 C0            LDA   ZEROFL       ; determine if the number is 00000
C05F  D0 05               BNE   EXIT         ; if not, then return
C061  A9 30               LDA   #48          ; print a zero
C063  20 D2 FF            JSR   CHROUT
C066  60         EXIT     RTS                ; we're finished
                                             ;
C067  01 00 0A   TB2SUB   .WORD 1,10,100,1000,10000
                                             ; two-byte table of subtrahends
C071  42 41 53   STRING   .ASC  "BASIC STARTS AT "
C081  00                  .BYTE 0
C082  00         ZEROFL   .BYTE 0            ; flag for first nonzero digit
```

See also BYTASC, FACPRD, FACPRT, NUMOUT.

Name

Convert a two-byte value to a floating-point number, using a ROM routine

Description

If you find occasion to use the built-in floating-point routines for trigonometric and other functions, this ROM routine is helpful. It converts a two-byte integer to its floating-point equivalent.

Prototype

1. JSR to GIVAYF with the low byte in .Y and the high byte in .A.
2. The result is returned in the floating-point accumulator.

Explanation

The GIVAYF routine is located at $8391 on the 64; $AF03 on the 128. (Be sure your program is operating with bank 15 in place before you call this routine on the 128.) The floating-point accumulator comprises locations $61–$66 on the 64; $63–$68 on the 128.

Routine

```
C000                GIVAYF    =     $B391     ; GIVAYF = $AF03 on the 128—ROM
                                              ; routine that converts into FP
                                              ;
C000  A9 32         MAIN      LDA   #50       ; the number 50 will be converted
C002  20 06 C0                JSR   CNVBFP    ; convert it
C005  60                      RTS
                                              ;
C006  A8            CNVBFP    TAY             ; the low byte goes into .Y
C007  A9 00                   LDA   #0        ; the high byte into .A
C009  20 91 B3                JSR   GIVAYF    ; the result is stored into FP accumulator at
                                              ; $61-$66 ($63-$68 on the 128)
C00C  60                      RTS
```

See also B2SNIN, B2UNIN, BCD2BY, CB2BCD, CFP2I, CI2FP.

Name

Character conversion using a lookup table

Description

Most of the routines in this book that convert one character code to another (for instance, from Commodore ASCII to screen codes) rely on the fact that ranges of characters frequently possess similar bit patterns. In these routines, you determine what range the character is in, usually by comparison with the low and high limits of the range. Based on the result, certain bitwise manipulations are carried out to complete the conversion.

This method works on most occasions. However, if you're faced with a situation in which you have to completely rearrange the order of the characters, and no ostensible bit patterns exist, you'll have to take another approach.

The **CNVERT** routine routine addresses that problem. At the same time, it offers a method of character conversion that is much faster than the others. And speed may be a requirement of your conversion routine, especially if the routine is incorporated into a terminal program where timing can be critical.

CNVERT itself is a very simple routine. It accepts an input character from the accumulator and, based on its number, returns the equivalent code from a lookup table at the end of your program. A one-to-one correspondence exists between the incoming and outgoing values. If the accumulator contains a 78 coming into the **CNVERT** routine, the seventy-eighth character value in the table is returned in .A.

The lookup table must be created beforehand. It can be built by the program using a conversion routine (as is done below) if the table follows a discernible pattern. Otherwise, it can be set up as a list of .BYTE statements.

Prototype

1. Transfer the incoming character value in .A to .Y.
2. Load the corresponding character value from the table as indexed by .Y and return.

Explanation

The example program first prepares a table of equivalent screen codes for all incoming Commodore ASCII characters in the routine TABPRE. This table (simply called TABLE here) is prepared by putting each Commodore ASCII value sequen-

tially through the conversion routine **CASSCR** and storing the value returned into the table. Since 256 characters are to be converted, the table itself is 256 bytes long. It's conveniently placed outside the working code at the end of the program.

After the lookup table has been created, the program accepts character values entered from the keyboard. Each character you type in is printed at the beginning of the screen, converted with **CNVERT** to the equivalent screen code, and POKEd to the screen, working back from the end of screen line 3. This continues until you type RETURN.

Routine

```
C000                CHROUT   =    65490
C000                GETIN    =    65508
C000                ZP       =    251
C000                SCREEN   =    1024        ; start of text screen
C000                COLRAM   =    55296       ; start of color RAM
C000                BGCOL0   =    53281       ; screen background color
C000                COLOR    =    646         ; COLOR = 241 on the 128
C000                BLACK    =    0
C000                MDGRAY   =    12
C000                PURPLE   =    4
                                              ;
                                              ; Input Commodore ASCII characters
                                              ; Convert to screen codes using a table
                                              ; and POKE resulting codes to the screen.
                                              ; Quit on RETURN.
C000                MAIN     =    *
C000  A9 93         CLRCHR   LDA  #147        ; clear the screen
C002  20 D2 FF               JSR  CHROUT
C005  A9 0C         BCKCOL   LDA  #MDGRAY     ; set screen background color to medium gray
C007  8D 21 D0               STA  BGCOL0
C00A  A9 04         TXTCOL   LDA  #PURPLE     ; set text color to purple
C00C  8D 86 02               STA  COLOR
C00F  20 36 C0               JSR  TABPRE      ; prepare conversion table
C012  A2 78                  LDX  #120        ; as an offset for POKEing screen codes
C014  CA            PRTLOP   DEX              ; position screen pointer for next character
C015  8E 8B C0               STX  TEMPX       ; save X since GETIN corrupts it
C018  20 E4 FF      WAIT     JSR  GETIN       ; get a character to convert
C01B  F0 FB                  BEQ  WAIT        ; if no character, wait
C01D  20 D2 FF               JSR  CHROUT      ; print Commodore ASCII character at start of
                                              ; screen
C020  C9 0D                  CMP  #13         ; is it RETURN?
C022  F0 11                  BEQ  FINISH      ; yes, so leave
C024  20 4D C0               JSR  CNVERT      ; use table to determine corresponding screen
                                              ; code
C027  AE 8B C0               LDX  TEMPX       ; restore X
C02A  9D 00 04               STA  SCREEN,X    ; store screen code at end of screen line 3 and
                                              ; work back
C02D  A9 00                  LDA  #BLACK      ; set foreground color of character to black
                                              ; (for early 64s)
C02F  9D 00 D8               STA  COLRAM,X
C032  4C 14 C0               JMP  PRTLOP      ; always continue printing
C035  60            FINISH   RTS
                                              ;
                                              ; TABPRE converts entire character set from
                                              ; Commodore ASCII to screen codes
C036  A0 00         TABPRE   LDY  #0          ; as an index
```

```
C038  8C 8C C0            STY   TEMPY         ; in case the conversion routine corrupts .Y.
C03B  AD 8C C0  TABLOP    LDA   TEMPY         ; counter for character number
C03E  20 52 C0            JSR   CASSCR        ; convert it to a screen code
C041  AC 8C C0            LDY   TEMPY         ; restore .Y
C044  99 8D C0            STA   TABLE,Y       ; store converted character to a screen code
                                              ; table
C047  EE 8C C0            INC   TEMPY         ; to convert next Commodore ASCII character
C04A  D0 EF               BNE   TABLOP        ; if we haven't done the entire set
C04C  60                  RTS                 ; return to MAIN
                                              ;
                                              ; Convert a Commodore ASCII value using
                                              ; the created lookup table.
C04D  A8        CNVERT    TAY                 ; character initially is in .A
C04E  B9 8D C0            LDA   TABLE,Y       ; look up corresponding screen code
C051  60                  RTS                 ; return to MAIN
                                              ;
                                              ; Convert Commodore ASCII in .A to screen
                                              ; code in .A.
                                              ; Upon returning, carry is clear.
                                              ; If no corresponding screen code exists, carry
                                              ; is set to indicate error and .A is the same.
C052  C9 FF     CASSCR    CMP   #255          ; is it pi?
C054  D0 04               BNE   NEQUIV        ; if not, check for nonequivalent codes
C056  A9 7E               LDA   #126          ; 255 becomes 126
C058  18                  CLC
C059  60                  RTS                 ; and we exit
C05A  8D 8A C0  NEQUIV    STA   TEMPA         ; preserve Commodore ASCII value for later
                                              ; checks
C05D  29 60               AND   #%01100000    ; check for nonequivalent codes (0-31 and
                                              ; 128-159)
C05F  D0 05               BNE   UPPLOW        ; if no, check for upper/lower half of
                                              ; character set
C061  AD 8A C0  ERROR     LDA   TEMPA         ; otherwise, no equivalent code
                                              ; Restore .A
C064  38                  SEC                 ; and indicate error
C065  60                  RTS
C066  AD 8A C0  UPPLOW    LDA   TEMPA         ; restore .A
C069  30 06               BMI   REMAIN
C06B  29 60               AND   #%01100000    ; in lower half
                                              ; First check whether in range 96-127.
C06D  C9 60               CMP   #%01100000    ; bit 5 and 6 are set if in 96-127
C06F  F0 12               BEQ   TOPLOW        ; if so, convert
                                              ;
                                              ; Otherwise, handle remainder (32-63, 64-95,
                                              ; 160-191, 192-223, 224-254)
                                              ; Shift bit 7 to 6 of TEMPA (containing the
                                              ; character) and set bit 7 to 0
C071  0E 8A C0  REMAIN    ASL   TEMPA         ; bit 7 of TEMPA into carry
C074  2A                  ROL                 ; carry into bit 0 of .A
C075  2E 8A C0            ROL   TEMPA         ; bit 6 of original TEMPA goes into carry
C078  6A                  ROR                 ; bit 0 of .A back into carry
C079  6E 8A C0            ROR   TEMPA         ; carry into bit 7
C07C  4E 8A C0            LSR   TEMPA         ; move 7 to 6 while setting 7 to 0
```

```
C07F  AD 8A C0          LDA   TEMPA         ; restore .A
C082  60                RTS                 ; and return (the LSR cleared the carry flag)
C083  AD 8A C0  TOPLOW  LDA   TEMPA         ; convert range 96-127
C086  29 5F             AND   #%01011111
C088  18                CLC                 ; and return with an equivalent code
C089  60                RTS
C08A  00        TEMPA   .BYTE0              ; for temporary .A storage
C08B  00        TEMPX   .BYTE0              ; for temporary .X storage
C08C  00        TEMPY   .BYTE0              ; for temporary .Y storage
C08D            TABLE   =     *             ; screen code table
C18D                    *=    *+256
```

See also CASSCR, CASTAS, SCRCAS, TASCAS, MIXLOW, MIXUPP, SWITCH.

Name

Cold start

Description

When you cold start the 64 or 128, the power-on reset routine causes the computer to go through certain initialization processes, just as when you first turn it on. On the 128, the MMU configuration registers are restored to their default settings, placing you in bank 15.

On both machines, the system ROMs are enabled (thus, you're returned to the regular character set if redefined characters are being used). If an autostart cartridge is in place on the 64, the cartridge cold-start vector at 32768 is executed. Otherwise, a RAM test is performed on both computers, and the 16 page-3 RAM vectors are restored. These include the interrupt vectors as well as a number of important Kernal I/O vectors. The computer also initializes the VIC-II chip (thereby restoring the default screen) and exits into the main BASIC loop, clearing the screen and printing the power-on message about BASIC and the number of bytes available.

In the process, the pointers to the BASIC program text are set to their default values. In effect, a BASIC NEW has been performed.

As you can see, then, performing a cold start has a dramatic effect on the computer. But, it's also ideal if you want to return the computer to its default condition when you exit your ML program.

Prototype

Jump to the power-on reset routine.

Explanation

The example program causes a cold start when the left-arrow key (in the upper left corner of the keyboard) is pressed.

COLDST itself is simple. It jumps to the cold-start routine in your computer. On the 64, this routine starts at 64738; on the 128, it's located at 65341.

Routine

```
C000              GETIN   =    65508
C000              RESET   =    64738      ; RESET = 65341 on the 128
                                          ;
                                          ; Perform a machine cold start with
                                          ; left arrow
                                          ; key.
C000 20 E4 FF LOOP        JSR  GETIN      ; get a character
C003 F0 FB                BEQ  LOOP       ; if no input
C005 C9 5F                CMP  #95        ; is it left-arrow character?
C007 D0 F7                BNE  LOOP       ; if not, get another key
C009 4C 0C C0             JMP  COLDST     ; execute cold start
                                          ;
                                          ; COLDST resets the computer.
C00C 4C E2 FC COLDST      JMP  RESET      ; cold start the computer
```

See also WARMST.

Name

Fill text screen color memory

Description

If you print characters to the screen, they will appear in the current cursor color. But if you store them to screen memory, characters will appear in the color currently in the corresponding color RAM position. With **COLFIL**, you can unify the overall text screen color by filling color RAM with one of the 16 colors.

The table gives the color values available on the 64 and 128 (40-column screen) and the colors they represent.

Color Values

Color Number	Color	Color Number	Color
0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light red
3	Cyan	11	Dark gray
4	Purple	12	Medium gray
5	Green	13	Light green
6	Blue	14	Light blue
7	Yellow	15	Light gray

Prototype

1. Enter this routine with the designated color value in .A.
2. Within a loop, fill all 1000 bytes of color RAM.

Explanation

The example program fills text screen color memory with purple, assigned as COLVAL.

Note: Another method of filling color memory, which requires less code, may be useful to you, depending on the version of ROM in your 64. Clearing the screen with CHR$(147) (see **CLRCHR** and **CLRROM**) affects screen color memory differently on different 64s. The earliest version of ROM (version 1) always fills color memory with white when the screen is cleared. With version 2, color memory is filled with the background color of the screen prior to the clear. So, to fill color memory with a particular color, you would simply store

your color value in the background color register at 53281 and clear the screen by printing CHR$(147). Then you would change the background to the color you prefer.

The most recent version of 64 ROM (version 3), and also 128 ROM, causes color memory to fill with the current cursor color when the screen is cleared. In this case, to fill color memory with a particular color, you would store the appropriate color value in the foreground text color register at 646 (241 on the 128) and clear the screen as before.

Routine

```
C000                COLRAM  =     55296          ; text screen color RAM location
                                                 ;
                                                 ; Fill color RAM with purple
C000  AD 18 C0              LDA   COLVAL         , get a color
C003  4C 06 C0              JMP   COLFIL         ; fill color RAM and RTS
                                                 ;
                                                 ; Fill text screen color RAM with color value
                                                 ; in .A.
C006  A0 FA         COLFIL  LDY   #250
C008  88            LOOP    DEY
C009  99 00 D8              STA   COLRAM,Y       ; 1st quarter
C00C  99 FA D8              STA   COLRAM+250,Y   ; 2nd quarter
C00F  99 F4 D9              STA   COLRAM+500,Y   ; 3rd quarter
C012  99 EE DA              STA   COLRAM+750,Y   ; 4th quarter
C015  D0 F1                 BNE   LOOP           ; all 250 bytes?
C017  60                    RTS
                                                 ;
C018  04            COLVAL  .BYTE 4              ; color purple
```

See also BCKCOL, BORCOL, TXTCCH, TXTCOL.

Name

Concatenate two files

Description

At times you may want to append the contents of one file to the end of a second file. That's what this routine does. Both of the original files remain unchanged; the new (third) file will contain a combination of the two original files.

Prototype

1. Open the disk command channel (Kernal SETLFS, SETNAM, OPEN).
2. Send the copy command as part of the SETNAM routine.
3. Close the command channel.

Explanation

This routine is basically the same as the **COPYFL** routine; however, instead of copying one file to another, you copy two files into a new file.

The filenames in the example are ABC and DEF, which are contained in the string that starts at $C01E. Note that they're separated by commas. What happens is that ABC is copied to a new file, followed by DEF. The result is a new, concatenated file called NEWFILE on disk.

Note: CONCAT will combine two sequential (SEQ) files just fine. If you try to concatenate two program (PRG) files, and then load the resulting program, only the first program will list. At the end of a program in memory are three zeros. When the LIST command finds the zeros, it stops. The second program is there, but it's just beyond the zeros and can't be accessed unless you go in and remove the final two zeros (and move the second part of the program down by two bytes).

Routine

```
C000                   SETLFS   =    $FFBA
C000                   SETNAM   =    $FFBD
C000                   OPEN     =    $FFC0
C000                   CLOSE    =    $FFC3
C000                   CLRCHN   =    $FFCC
                                                ;
C000  A9  01           CONCAT   LDA  #1         ; logical file (1)
C002  A2  08                    LDX  #8         ; disk drive is device 8
C004  A0  [illegible]           LDY  #15        ; command channel 15
C006  20  BA  FF                JSR  SETLFS     ; prepare to open it
C009  A9  17                    LDA  #BUFLEN    ; length of buffer
C00B  A2  1E                    LDX  #<BUFFER   ; X and Y hold the
```

```
C00D  A0  C0               LDY   #>BUFFER     ; address of the buffer
C00F  20  BD  FF           JSR   SETNAM       ; set name
C012  20  C0  FF           JSR   OPEN         ; open it
C015  A9  01               LDA   #1           ; and immediately
C017  20  C3  FF           JSR   CLOSE        ; close the command channel
C01A  20  CC  FF           JSR   CLRCHN       ; clear the channels
C01D  60                   RTS                ; all done
                                              ;
                                              ; Data area
C01F  43  30  3A  BUFFER   .ASC  "C0:NEWFILE=0:ABC,0:DEF"
                                              ; substitute your own filenames
C034  0D                   .BYTE 13           ; RETURN character
C035              BUFLEN   =     * - BUFFER
```

See also COPYFL, FORMAT, INITLZ, RENAME, SCRTCH, VALIDT.

Name

Copy a file to the same disk

Description

The DOS Copy command is really intended for making backups with a dual drive, but Commodore hasn't manufactured a dual drive for several years. Thus, the copy command is useful only for copying a file (under a different name) to the disk it already occupies.

Prototype

1. Open channel 15 (Kernal routines SETLFS, SETNAM, OPEN).
2. As part of the name, include the copy command.
3. Close the command channel.

Explanation

The key to this routine is the string at the end of the program, "C0:NEWFILE=0:OLDFILE", which tells the disk drive to copy the program OLDFILE on drive 0 to the file named NEWFILE on the same drive.

The SETLFS routine sets up logical file 1, drive 8, channel 15. Then SETNAM sets the length and address of the command and we OPEN. Then, the job finished, we close the channel.

In actual practice, you may want to set up a separate buffer for the copy command and write different parameters to the data area. After all, it's fairly rare that you'll always be copying files called OLDFILE to a new name called NEWFILE.

Note: If you own additional disk drives, you may want to change the drive number at \$C002–\$C003 to 9, 10, or 11. Also, if you own a dual drive, you may change one or both of the zeros in the ASCII string to ones.

Routine

```
C000                SETLFS   =    $FFBA
C000                SETNAM   =    $FFBD
C000                OPEN     =    $FFC0
C000                CLOSE    =    $FFC3
C000                CLRCHN   =    $FFCC
                                                 ;
C000  A9 01         COPYFL   LDA  #1             ; logical file (1)
C002  A2 08                  LDX  #8             ; disk drive is device 8
C004  A0 0F                  LDY  #15            ; command channel 15
C006  20 BA FF               JSR  SETLFS         ; prepare to open it
C009  A9 15                  LDA  #BUFLEN        ; length of buffer
C00B  A2 1E                  LDX  #<BUFFER       ; .X and .Y hold the
C00D  A0 C0                  LDY  #>BUFFER       ; address of the buffer
```

```
C00F  20  BD  FF              JSR    SETNAM      ; set name
C012  20  C0  FF              JSR    OPEN        ; open it
C015  A9  01                  LDA    #1          ; and immediately
C017  20  C3  FF              JSR    CLOSE       ; close the command channel
C01A  20  CC  FF              JSR    CLRCHN      ; clear the channels
C01D  60                      RTS                ; all done
                                                 ; Data area
C01E  43  30  3A  BUFFER      .ASC   "C0:NEWFILE=0:OLDFILE"
                                                 ; substitute your own filenames
C032  0D                      .BYTE  13          ; RETURN character
C033              BUFLEN      =      * - BUFFER
```

See also CONCAT, FORMAT, INITLZ, RENAME, SCRTCH, VALIDT.

Name

Custom characters for the 80-column screen

Description

Using the routine that writes to the 128's 80-column chip, **CUST80** redefines one character. This routine can easily be expanded to create an entirely new character set.

Prototype

1. Set up registers 18 and 19 of the VDC chip to point to the address of the letter *A* (uppercase/graphics mode).
2. Send eight bytes to register 31 to create the new character.

Explanation

The key to accessing the 80-column VDC chip is writing to locations $D600 and $D601, the gateway bytes (see **RE80CO** and **WR80CO** for more about the gateway bytes). The STRVDC routine at $0C26 below handles this task. First, the VDC register to be POKEd is stored in $D600. Next, we need to wait for bit 7 of $D600 to turn on. At that point, $D601 can be PEEKed or POKEd.

The VDC's uppercase/graphics character set starts at location $2000 within the VDC's private 16K of memory. The shape for the letter *A* is found at $2010. So, to change that shape, the routine must set up the address $2010 in registers 18 and 19. Note that, unlike most other addresses in the 128, in this case the high byte is stored ahead of the low byte. (This could be called a quirk of the VDC.) STRVDC is called twice—once to store a $20 into register 18, and once to store a $10 into 19.

When the POKE address has been established, the values to be sent there are stored in VDC register 31. The 80-column chip automatically increments the address, so it's not necessary to keep writing to registers 18 and 19. The character shape in the source code is stored in binary form, so the actual appearance can be seen. The letter *A* is replaced by a small *z* inside a box.

The character sets are stored in a rather unusual fashion. The first eight bytes ($2000–$2007) are the @ character. The next eight bytes are unused. The next eight ($2010–$2017) are

the letter *A,* followed by eight more unused bytes. This pattern continues. If you're planning to store several consecutive custom characters, remember to skip eight bytes between shapes.

Note: Both character sets can be displayed at the same time. Attribute memory determines which set is used. (See **VDCCOL** for more information about attribute memory.) The second half of each character set contains the reversed versions of the first 128 characters. These characters are what you see when you turn reverse mode on. Now, attribute memory can be changed to display a normal or a reverse character (again, see **VDCCOL**), which means that the reverse character shapes in the character set are redundant. It is actually possible to have four character sets in memory at the same time, a total of 512 characters. To reverse any of them, write to attribute memory (which gives you 512 more, reversed characters).

Routine

```
0C00                VDCADR   =     $D600
0C00                VDCDAT   =     $D601
0C00                VRMLO    =     19
0C00                VRMHI    =     18           ; note the high byte is first, not second
0C00                VRDAT    =     31
0C00                MEM4A    =     $2010        ; (internal memory for the VDC)
                                                ;
0C00  A9 [illegible]  CUST80   LDA   #>MEM4A      ; high byte of character memory
0C02  A2 12                  LDX   #VRMHI       ; register 18
0C04  20 26 0C               JSR   STRVDC       ; set up the register
0C07  A9 10                  LDA   #<MEM4A      ; low byte
0C09  A2 13                  LDX   #VRMLO       ; register 19
0C0B  20 26 0C               JSR   STRVDC       ; and store the value
                                                ;
0C0E  A0 00                  LDY   #0
0C10  B9 1E 0C      LOOP     LDA   CHAR,Y
0C13  A2 1F                  LDX   #VRDAT       ; register 31
0C15  20 [illegible] 0C      JSR   STRVDC       ; store it
0C18  C8                     INY                ; we have to move forward
0C19  C0 08                  CPY   #8
0C1B  D0 F3                  BNE   LOOP
0C1D  60                     RTS                ; done
                                                ;
0C1E                CHAR     =     *
0C1E  FF                     .BYTE %11111111
0C1F  81                     .BYTE %10000001
0C20  B5                     .BYTE %10110101
0C21  89                     .BYTE %10001001
0C22  91                     .BYTE %10010001
0C23  AD                     .BYTE %10101101
0C24  81                     .BYTE %10000001
0C25  FF                     .BYTE %11111111
                                                ;
```

```
0C26                STRVDC   =     *
0C26  8E  00  D6             STX   VDCADR     ; store .X in the address gate
0C29  AE  00  D6    WAITAD   LDX   VDCADR     ; and wait
0C2C  10  FB                 BPL   WAITAD     ; for bit 7 to click
0C2E  8D  01  D6             STA   VDCDAT     ; store the data
0C31  60                     RTS              ; and quit
```

See also ANIMAT, CHRDEF, RE80CO, VDCCOL, WR80CO.

Name

Create DATA statements from numbers in memory

Description

If you have a short ML program—or sprites, custom characters, or other chunk of memory—you wish to add to a BASIC program, this program will convert the values in memory to a series of DATA statements that are tacked onto the end of the program currently in memory.

Prototype

1. Enter with the starting address in DFIRST and the ending address (plus one) in DLAST.
2. Subtract 2 from the pointer to the end of BASIC text and store this pointer in zero-page.
3. Begin a BASIC line by storing two bogus nonzero line links, which will be fixed later.
4. Next, store a two-byte line number (data from memory location 49152 will be put in line 49152, for example) and the BASIC token that represents the keyword DATA.
5. Loop six times, reading a byte from memory and converting it to ASCII characters.
6. If the loop isn't finished, add a comma between numbers.
7. After each line, store a zero-byte and go back to step 3.
8. When the last byte is converted, call the ROM routine LINKPRG to fix the line links.

Explanation

Before you SYS or JSR to this routine, store the beginning address in DFIRST and the ending address (plus one) in DLAST. For example, to create DATA statements for the range 8192–16191, you would put an 8192 in DFIRST, but a 16192 (one byte past 16191) in DLAST.

BASIC program lines have an overhead of five bytes, four at the beginning and one at the end. The first two are the line link, which points to the line link of the next BASIC line (the final link is two zeros, which mark the end of the program). After the link comes the line number, low-byte first. At the end of each line you'll find a zero byte.

To manufacture DATA statements, we put two nonzero numbers into the line-link area, and then a line number. The example program numbers the lines according to where in memory they're stored. So line 16394 would mark the beginning of the bytes that go into memory at 16394. After the line

link and the line number, an $83 is stored. This is the BASIC token for DATA.

The values from memory are changed to ASCII in the subroutine called ASCII. The number 153 would be converted to the three characters 1, 5, and 3. It's similar to the **BYTASC** routine elsewhere in this book. Between the numbers, commas are stored.

Routine

```
C000                ZP       =    $FB
C000                VARTAB   =    45           ; replace with TXTTOP = 4624 for the 128
C000                LINKPRG  =    $A533        ; LINKPRG = $4F4F on 128
                                               ;
C000  AD E7  C0  DATAMK   LDA   DFIRST         ; low byte of beginning of memory to
                                               ; convert
C003  8D 27  C0           STA   POINTR         ; into POINTR below
C006  AD E8  C0           LDA   DFIRST+1       ; high byte
C009  8D 28  C0           STA   POINTR+1       ; also
C00C  A5 2D               LDA   VARTAB         ; get the end-of-BASIC pointer (substitute
                                               ; TXTTOP for the 128)
C00E  38                  SEC
C00F  E9 02               SBC   #2             ; subtract 2
C011  85 FB               STA   ZP             ; save it in ZP
C013  A5 2E               LDA   VARTAB+1       ; high byte (substitute TXTTOP+1 for the
                                               ; 128)
C015  E9 00               SBC   #0             ; subtract zero, to account for page
                                               ; boundaries
C017  85 FC               STA   ZP+1
C019  20 7B  C0  NEWLIN   JSR   BOGUS          ; set up a false line link
C01C  20 86  C0           JSR   LINNUM         ; create the line number and data token
C01F  A9 06               LDA   #6             ; number of data numbers per line
C021  8D EB  C0           STA   NUMDAT         ; save it
C024  A0 00      MORELN   LDY   #0
C026  B9 FF  FF  LOADR    LDA   $FFFF,Y        ; this will be fixed
C029             POINTR   =     LOADR+1        ; self-modifying code
C029  20 9C  C0           JSR   ASCII          ; make into ASCII numbers and store in
                                               ; memory
C02C  EE 27  C0           INC   POINTR         ; add one to POINTR
C02F  D0 03               BNE   NOHI
C031  EE 28  C0           INC   POINTR+1
C034  AD 28  C0  NOHI     LDA   POINTR+1       ; see if we're done
C037  CD EA  C0           CMP   DLAST+1        ; does it equal the last byte?
C03A  F0 11               BEQ   LOOKLO         ; maybe, look at the low byte
C03C  CE EB  C0  ANDER    DEC   NUMDAT         ; count down (six numbers per line)
C03F  F0 2F               BEQ   ENDLIN         ; fix the end of the line
C041  A9 2C               LDA   #44            ; else insert a comma
C043  A0 00               LDY   #0
C045  91 FB               STA   (ZP),Y         ; store in memory
C047  20 E0  C0           JSR   PLUSZP         ; add to ZP
C04A  4C 24  C0           JMP   MORELN         ; go back for another byte from memory
C04D  AD 27  C0  LOOKLO   LDA   POINTR         ; check the low byte
C050  CD E9  C0           CMP   DLAST          ; against DLAST
C053  D0 E7               BNE   ANDER          ; not equal, do more
                                               ;
                                               ; Clean up the end of the program.
C055  A9 00               LDA   #0
C057  A0 02               LDY   #2
C059  91 FB      CLNLP    STA   (ZP),Y         ; put three zeros at the end of the program
C05B  88                  DEY
C05C  10 FB               BPL   CLNLP
```

```
C05E  20 DD C0            JSR   PL2ZP        ; double INC ZP
C061  20 E0 C0            JSR   PLUSZP       ; one more time
C064  A5 FB               LDA   ZP           ; set end-of-program pointer
C066  85 2D               STA   VARTAB       ; (substitute TXTTOP for the 128)
C068  A5 FC               LDA   ZP+1
C06A  85 2E               STA   VARTAB+1     ; (substitute TXTTOP for the 128)
C06C  20 33 A5            JSR   LINKPRG      ; relink the lines
C06F  60                  RTS                ; that's it
                                             ;
C070  A9 00     ENDLIN    LDA   #0           ; put a zero
C072  A8                  TAY                ; at the end of the line
C073  91 FB               STA   (ZP),Y       , store it
C075  20 E0 C0            JSR   PLUSZP       ; move ZP up one
C078  4C 19 C0            JMP   NEWLIN
C07B  A9 01     BOGUS     LDA   #1           ; put ones in the line links, to be fixed
                                             ; later
C07D  A8                  TAY
C07E  91 FB     BOGLP     STA   (ZP),Y
C080  88                  DEY
C081  10 FB               BPL   BOGLP
C083  4C DD C0            JMP   PL2ZP        ; double INC the ZP pointer
                                             ;
C086  A0 01     LINNUM    LDY   #1           ; copy the memory address to the line
                                             ; number
C088  B9 27 C0  LINLP     LDA   POINTR,Y
C08B  91 FB               STA   (ZP),Y
C08D  88                  DEY
C08E  10 F8               BPL   LINLP
C090  20 DD C0            JSR   PL2ZP
C093  A0 00               LDY   #0
C095  A9 83               LDA   #$83         ; token for the data command
C097  91 FB               STA   (ZP),Y
C099  4C E0 C0            JMP   PLUSZP
                                             ;
C09C  AA        ASCII     TAX                ; save in .X
C09D  C9 64               CMP   #100         ; is it smaller than 100?
C09F  B0 06               BCS   HAGHUN       ; no, do a hundreds place
C0A1  C9 0A               CMP   #10          ; less than 100; is it less than 10?
C0A3  B0 14               BCS   TENS         ; no, so it has a tens place
C0A5  90 23               BCC   ONES         ; it is less than 10; go to ONES
C0A7  A0 31     HAGHUN    LDY   #49          ; put an ASCII 1 in .Y
C0A9  20 D1 C0            JSR   MIN100       ; subtract 100
C0AC  C9 64               CMP   #100         ; is it still higher than 100?
C0AE  90 04               BCC   STORHN       ; no, continue
C0B0  C8                  INY                ; yes
C0B1  20 D1 C0            JSR   MIN100       ; so subtract again
C0B4  AA        STORHN    TAX                ; save in .X
C0B5  98                  TYA                ; put an ASCII 1 or 2 into .A
C0B6  20 D5 C0            JSR   PUTMEM
C0B9  8A        TENS      TXA                ; get the number back
C0BA  A0 30               LDY   #48
C0BC  C9 0A     COM10     CMP   #10          ; compare .A to 10
C0BE  90 05               BCC   HAGTEN       ; get ready to leave
C0C0  E9 0A               SBC   #10          ; subtract 10
C0C2  C8                  INY                ; .Y increases
C0C3  D0 F7               BNE   COM10        ; branch always
C0C5  AA        HAGTEN    TAX                ;
C0C6  98                  TYA
C0C7  20 D5 C0            JSR   PUTMEM
                                             ;
C0CA  8A        ONES      TXA
C0CB  09 30               ORA   #48
```

```
C0CD 20  D5  C0          JSR   PUTMEM
C0D0 60                  RTS
                                              ;
C0D1 38          MIN100  SEC
C0D2 E9  64              SBC   #100
C0D4 60                  RTS
                                              ;
C0D5 A0  00      PUTMEM  LDY   #0
C0D7 91  FB              STA   (ZP),Y         ; and store it
C0D9 20  E0  C0          JSR   PLUSZP
C0DC 60                  RTS
                                              ;
C0DD 20  E0  C0  PL2ZP   JSR   PLUSZP
C0E0 E6  FB      PLUSZP  INC   ZP             ; INC ZP by one
C0E2 D0  02              BNE   FINZP          ; if not equal, end
C0E4 E6  FC              INC   ZP+1           ; else, add one to high byte
C0E6 60          FINZP   RTS
                                              ;
C0E7 00  C0      DFIRST  .WORD$C000
C0E9 0A  C0      DLAST   .WORD$C00A
C0EB 00          NUMDAT  .BYTE 0
```

See also RENUM1.

Name

Check the disk status and print a message

Description

DERRCK reads the disk drive's error channel and looks for certain common problems. For example, if you try to write to a disk that has a write-protect tab, an error 26 will result. When an error 26 is discovered, **DERRCK** prints a message that says *Please remove write-protect tab.*

Prototype

1. In preparation for **DERRCK**, open the command channel (15,8,15).
2. Within **DERRCK**, first print the message *DISK STATUS:*.
3. Read the error channel (using the Kernal routines CHKIN and CHRIN) and print the characters received.
4. Convert the error number to a binary coded decimal (BCD) number as it's received.
5. Search through a table of specific errors.
6. If the error number matches a number in the table, print a message that provides more information.

Explanation

The example routine attempts to open a file that doesn't exist on the disk. The **DERRCK** routine then reads the error channel and prints the message *Filename doesn't exist on disk, try again.*

The Kernal routines SETLFS, SETNAM, and OPEN should be called early in the program. **DERRCK** performs a Kernal CHKIN to cause input to come from channel 15 instead of the keyboard. The PRINTS subroutine is a general string-printing routine. The first thing it prints is the DISK STATUS: line. Next, the error channel is read and printed. The error number comes in as two ASCII numbers; error 73 would appear as two characters ($37 and $33). The ASCII numbers are combined into one byte ($73, in this case) to make looking up the error a little easier.

Several error numbers can be ignored (0–20, 50, and 73). Others are fairly common (26, 33, 74, and 62). When one of the four common errors is encountered, a longer message is printed, again via PRINTS.

Routine

```
C000                 ZP       =    $FB
C000                 SETLFS   =    $FFBA
C000                 SETNAM   =    $FFBD
C000                 OPEN     =    $FFC0
C000                 CHKIN    =    $FFC6
C000                 CLOSE    =    $FFC3
C000                 CHRIN    =    $FFCF
C000                 CHROUT   =    $FFD2
C000                 READST   =    $FFB7
C000                 CLRCHN   =    $FFCC
                                                  ;
C000  A9 0F                   LDA  #15            ; logical file
C002  A8                      TAY                 ; secondary address (command channel)
C003  A2 08                   LDX  #8             ; device number
C005  20 BA FF                JSR  SETLFS         ; get the channel ready
C008  A9 00                   LDA  #0             ; no filename
C00A  20 BD FF                JSR  SETNAM         ; set the name
C00D  20 C0 FF                JSR  OPEN           ; and open it
C010  A9 02                   LDA  #2             ; logical file
C012  A8                      TAY                 ; the secondary address
C013  A2 08                   LDX  #8             ; a disk file
C015  20 BA FF                JSR  SETLFS
C018  A9 0E                   LDA  #LEN           ; the length of the fake filename
C01A  A2 AB                   LDX  #<FAKE
C01C  A0 C0                   LDY  #>FAKE         ; address of fake
C01E  20 BD FF                JSR  SETNAM         ; this is not a file
C021  20 C0 FF                JSR  OPEN           ; open it (error now)
C024  20 32 C0                JSR  DERRCK         ; check the status
C027  A9 02                   LDA  #2
C029  20 C3 FF                JSR  CLOSE          ; close channel 2
C02C  A9 0F                   LDA  #15
C02F  20 C3 FF                JSR  CLOSE          ; close channel 15
C031  60                      RTS                 ; and finish
                                                  ;
C032  A2 0F          DERRCK   LDX  #15            ; logical file 15
C034  20 C6 FF                JSR  CHKIN          ; ready for input
C037  A2 B9                   LDX  #<DSTAT
C039  A0 C0                   LDY  #>DSTAT
C03B  20 97 C0                JSR  PRINTS         ; print the DSTAT message
C03E  20 CF FF                JSR  CHRIN          ; get the first number
C041  20 D2 FF                JSR  CHROUT
C044  0A                      ASL
C045  0A                      ASL
C046  0A                      ASL
C047  0A                      ASL                 ; shift it left four times
C048  8D AA C0                STA  ERROR          ; high nybble
C04B  20 CF FF                JSR  CHRIN          ; get the next one
C04E  20 D2 FF                JSR  CHROUT
C051  29 0F                   AND  #%00001111     ; mask out the high nybble
C053  0D AA C0                ORA  ERROR          ; add to ERROR
C056  8D AA C0                STA  ERROR          ; and store it
C059  20 CF FF       MORE     JSR  CHRIN          ; get a character from disk
C05C  C9 0D                   CMP  #13            ; is it a carriage return?
C05E  F0 06                   BEQ  EXAMIT         ; if so, we're done
C060  20 D2 FF                JSR  CHROUT         ; else print it
C063  4C 59 C0                JMP  MORE
C066  20 D2 FF       EXAMIT   JSR  CHROUT         ; print the carriage return
C069  AD AA C0                LDA  ERROR          ; get the error number
C06C  C9 21                   CMP  #$21           ; is it 0-20?
C06E  90 23                   BCC  ALLDONE        ; if so, exit
C070  A0 01                   LDY  #<OKNUM        ; check for OK errors
C072  D9 C7 C0       OKLOOP   CMP  OK,Y           ; if it matches
C075  F0 1C                   BEQ  ALLDONE        ; skip ahead
```

```
C077  88                      DEY
C078  10  F8                  BPL   OKLOOP        ; loop back
C07A  A0  03                  LDY   #<NOKNUM      ; the error is not OK
C07C  D9  C9  C0  NOKLOOP     CMP   NOK,Y         ; check NOK table
C07F  F0  03                  BEQ   MESSAGE       ; found it, so print a message
C081  88                      DEY
C082  10  F8                  BPL   NOKLOOP       ; loop back for more
                                                  ;
C084  98          MESSAGE     TYA                 ; index to .A
C085  0A                      ASL                 ; times 2
C086  A8                      TAY                 ; back in .Y
C087  B9  CD  C0              LDA   NTABLE,Y      ; find the low byte
C08A  AA                      TAX                 ; into .X
C08B  C8                      INY                 ; go up 1
C08C  B9  CD  C0              LDA   NTABLE,Y      ; high byte
C08F  A8                      TAY                 ; into .Y
C090  20  97  C0              JSR   PRINTS        ; print the message
C093  20  CC  FF  ALLDONE     JSR   CLRCHN        ; clear the channels
C096  60                      RTS                 ; and the subroutine is done
                                                  ;
C097  86  FB      PRINTS      STX   ZP            ; low byte in ZP
C099  84  FC                  STY   ZP+1          ; high byte, too
C09B  A0  00                  LDY   #0            ; get ready to print it
C09D  B1  FB      PSLOOP      LDA   (ZP),Y        ; get a character
C09F  48                      PHA                 ; push it
C0A0  20  D2  FF              JSR   CHROUT        ; print it
C0A3  C8                      INY
C0A4  68                      PLA                 ; pull it
C0A5  C9  0D                  CMP   #13           ; is it a RETURN?
C0A7  D0  F4                  BNE   PSLOOP        ; if not, get another character
C0A9  60                      RTS
                                                  ;
C0AA  00          ERROR       .BYTE 00
C0AB  30  3A  4E  FAKE        .ASC  "0:NOTAFILENAME"
C0B9              LEN         =     *-FAKE
C0B9  44  49  53  DSTAT       .ASC  "DISK STATUS: "
C0C6  0D                      .BYTE 13
;
C0C7  50  73      OK          .BYTE $50,$73
C0C9              OKNUM       =     *-OK-1        ; number of OK errors
C0C9  26  33  74  NOK         .BYTE $26,$33,$74,$62
C0CD              NOKNUM      =     *-NOK-1       ; number of not OK errors
C0CD  D5  C0  F6  NTABLE      .WORD WRPROT,WILDCD,NREADY,NFOUND
C0D5  50  4C  45  WRPROT      .ASC  "PLEASE REMOVE WRITE-PROTECT TAB."
C0F5  0D                      .BYTE 13
C0F6  4E  4F  20  WILDCD      .ASC  "NO *'S OR ?'S ALLOWED IN FILENAME."
C118  0D                      .BYTE 13
C119  50  4C  45  NREADY      .ASC  "PLEASE INSERT DISK OR TURN ON THE DRIVE."
C141  0D                      .BYTE 13
C142  46  49  4C  NFOUND      .ASC  "FILENAME DOESN'T EXIST ON DISK, TRY AGAIN."
C16C  0D                      .BYTE 13
```

See also CHK144, RDSTAT.

Name

Read the directory as a stream of bytes

Description

DIRBYT prints the directory on the screen without actually loading the directory file into memory (which is what **DIRPRG** does). Thus, any programs in the BASIC workspace are preserved.

Prototype

1. On the 128, set the bank to 15.
2. OPEN 1,8,0 with the name "$0" (SETLFS, SETNAM, and OPEN).
3. On the 128, prior to SETNAM, load .A with the bank where the directory is to be OPENed and .X with the bank containing the directory filename, then SETBNK.
4. Discard the two track and sector bytes.
5. Check the two link bytes for the last entry.
6. If they're both zeros, exit the routine.
7. Otherwise, get and print (with **NUMOUT**) the number of blocks in the current entry on a new screen line.
8. Get characters from the current entry and print them until a zero byte is reached.
9. If a zero byte is reached, loop back to step 5.
10. If the next set of link bytes are both zeros, close file 1 and restore default devices.

Explanation

DIRBYT reads the directory byte by byte and displays it in a formatted fashion on the text screen.

The directory file is structured just like a BASIC program file, which is why you can type LOAD "$0",8 and LIST it as if it were a program. At the beginning of the directory are two bytes that would indicate the load address if it really were a program. We have no use for these, and they are discarded.

The next two bytes are link bytes that point to the address in memory of the next entry in the file. These are equivalent to the link bytes in a BASIC program file that point to the next program line. If the two link bytes are both zeros (determined in CHLINK), we know we've reached the end of the file (likewise with a BASIC program). When this occurs, we branch to EXIT, closing file 1 and restoring default devices.

If one or both of the link bytes are nonzero bytes, we get and print characters from the current entry until a zero byte is

reached. A zero marks the end of a line, again just as in a BASIC program line.

Each entry can be one of three types: the disk name, a program name, or the BLOCKS FREE message. The first two bytes after the link bytes in each program entry represent the number of blocks occupied by the corresponding program on the disk. If the entry is the BLOCKS FREE message at the bottom of the directory, the first two bytes refer to the number of blocks remaining on the disk. If the entry is the disk name, the first two bytes are zeros.

Regardless of the entry type, these first two bytes are printed as a two-byte integer with **NUMOUT**, a space is inserted, and the rest of the entry printed (in LOOP).

As is suggested with **DIRPRG**, you can display a portion of the directory by using the built-in wildcard notations. For instance, to show all two-character filenames that begin with *D*, change the directory filename in FILENM to "$0:D?". Or to show any filename beginning with *D*, regardless of its length, change FILENM to "$0:D*".

Note: **DIRBYT** lacks disk error checking. You can easily add this feature if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM, as noted in the source listing. Jump to **DERRCK** immediately after you have opened file 1 to the disk. Also, as noted in the source listing, be sure to open the error channel (15) at the beginning of the program.

On the 128, include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000        SETLFS  =  65466
C000        SETBNK  =  65384      ; Kernal bank number for OPEN and
                                  ; filename (128 only)
C000        MMUREG  =  65280      ; MMU configuration register (128 only)
C000        SETNAM  =  65469
C000        OPEN    =  65472
C000        CHKIN   =  65478
C000        CHRIN   =  65487
C000        CHROUT  =  65490
C000        CLOSE   =  65475
C000        CLRCHN  =  65484
C000        ZP      =  251
C000        LINPRT  =  48589      ; LINPRT = 36402 on the 128
                                  ;
                                  ; Read the directory as a stream of bytes.
                                  ; Open channel 15 here if you include disk
                                  ; error checking (DERRCK).
                                  ;
```

```
C000                DIRBYT   =     *
                                         ; LDA #0; set the 128 to bank 15 (128 only)
                                         ; STA MMUREG; (128 only)
C000  A9 01                  LDA   #1        ; logical file 1
C002  A2 08                  LDX   #8        ; disk drive (sometimes device 9)
C004  A0 00                  LDY   #0        ; 1,8,0 is set for read
C006  20 BA FF               JSR   SETLFS    ; set parameters for read
                                         ;
                                         ; Include the following three instructions on
                                         ; the 128.
                                         ; LDA BNKNUM; open into bank number
                                         ; LDX BNKFNM; bank containing the ASCII
                                         ; filename
                                         ; JSR SETBNK
                                         ;
C009  A9 02                  LDA   #FNLENG   ; length of filename
C00B  A2 5D                  LDX   #<FILENM  ; the filename is "$0"
C00D  A0 C0                  LDY   #>FILENM
C00F  20 BD FF               JSR   SETNAM    ; set up filename
C012  20 C0 FF               JSR   OPEN      ; open the directory file for reading
                                         ;
                                         ; Insert JSR DERRCK here for disk error
                                         ; checking
                                         ;
C015  A2 01                  LDX   #1
C017  20 C6 FF               JSR   CHKIN     ; input from file 1
C01A  20 57 C0               JSR   GET2      ; discard the track and sector bytes
C01D  20 49 C0 NEWENT        JSR   CHLINK    ; is it the last entry?
C020  F0 1E                  BEQ   EXIT      ; if so, exit the routine
C022  A9 0D                  LDA   #13       ; print each entry on a new physical line
C024  20 D2 FF               JSR   CHROUT
                                         ; Get the number of blocks in the next
                                         ; entry and print with NUMOUT.
C027  20 CF FF               JSR   CHRIN     ; get the low byte
C02A  AA                     TAX             ; and put in .X
C02B  20 CF FF               JSR   CHRIN     ; get the high byte in .A
C02E  20 CD BD NUMOUT        JSR   LINPRT    ; print the number
C031  A9 20                  LDA   #32       ; insert a SPACE
C033  20 D2 FF               JSR   CHROUT
                                         ;
                                         ; Read information on each program entry
                                         ; (filename, type, etc.).
C036  20 CF FF LOOP          JSR   CHRIN     ; input a character from entry
C039  F0 E2                  BEQ   NEWENT    ; if zero byte, next byte is from a new entry
C03B  20 D2 FF               JSR   CHROUT    ; print it
C03E  D0 F6                  BNE   LOOP      ; and continue with current entry
C040  A9 01    EXIT          LDA   #1        ; you're finished
C042  20 C3 FF               JSR   CLOSE     ; close logical file 1
C045  20 CC FF               JSR   CLRCHN    ; clear all channels and restore default
                                         ; devices
C048  60                     RTS
                                         ;
C049  20 CF FF CHLINK        JSR   CHRIN     ; check two link bytes for 00
C04C  85 FB                  STA   ZP        ; store first byte
C04E  20 CF FF               JSR   CHRIN     ; get another byte
C051  05 FB                  ORA   ZP        ; and OR it with the first byte
C053  60                     RTS             ; return a zero if both are zero, otherwise
                                         ; nonzero value returned
                                         ;
C054  20 57 C0 GET4          JSR   GET2      ; get next four bytes from file 1
C057  20 CF FF GET2          JSR   CHRIN     ; get two bytes
```

```
C05A  4C  CF  FF                JMP    CHRIN       ; get a byte and RTS
                                                   ;
                                                   ; Insert DERRCK here if you're including
                                                   ; error checking.
                                                   ;
C05D  24  30         FILENM     .ASC   "$0"        ; filename for directory
C05F                 FNLENG     =      * FILENM    ; length of filename
                                                   ; Include the next two variables on the 128
                                                   ; BNKNUM  BYTE 0; bank number to OPEN
                                                   ; into (128 only)
                                                   ; BNKFNM  BYTE 0, bank number where
                                                   ; ASCII filename is (128 only)
```

See also DIRPRG, FRESEC.

Name

Load the directory as a program file

Description

This routine loads the directory file on disk into the BASIC workspace. If you've worked in BASIC, you've probably done this many times with LOAD"$",8. If so, you've certainly found, perhaps the hard way, that loading the directory in this manner overwrites any BASIC program currently in memory. But if the program you're executing is outside the BASIC workspace, which is often the case with ML, this method of reading the directory is completely suitable.

Prototype

1. On the 128, set the bank to 15.
2. Set up the parameters for a relative load of the directory file (SETLFS, SETNAM).
3. On the 128, prior to SETNAM, load .A with the bank where the directory is to be loaded and .X with the bank containing the directory filename. Then JSR to SETBNK.
4. Store zero in .A to indicate a load operation.
5. Load .X and .Y with the starting address of BASIC from TXTTAB.
6. JSR to LOAD.
7. Store .X and .Y in the end-of-BASIC text pointer.
8. JMP to LINKPG.

Explanation

DIRPRG loads the directory as a BASIC program into the current BASIC workspace. (A secondary address of zero causes a relative load.) This allows you to position the BASIC workspace anywhere you want before entering the routine. **DIRPRG** simply loads the directory file based on the current starting address of BASIC.

DIRPRG is very much like a relative load of any BASIC program (see **LOADBS**). As with **LOADBS**, we place a zero in the accumulator before executing the Kernal LOAD to cause a load rather than to verify. And again, before JSRing to LOAD, we store the starting address of BASIC (TXTTAB) in .X and .Y. (On the 128, TXTTAB is at location 45.)

After LOAD has finished, store .X and .Y containing the ending address of the directory file in VARTAB (or TEXTTP at 4624 on the 128). Finish up by JMPing to LINKPG to relink the lines of the directory file as a BASIC program.

Note: You can look at different portions of the directory selectively by using the operating system's built-in wildcard notations. For instance, if you want to display a list of all files whose names begin with *PROG*, change FILENM in **DIRPRG** to "$0:PROG*". On the other hand, if you want a list of all program names ending in *.OBJ* that are ten characters long, change FILENM to "$0:??????.OBJ−P".

DIRPRG currently lacks disk error checking. You can add this feature if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM, as noted in the source listing. Jump to **DERRCK** immediately after the JSR LOAD instruction. Be sure to open the error channel (15) at the beginning of the program (also noted in the source listing).

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000             SETLFS   =    65466
C000             SETNAM   =    65469
C000             LOAD     =    65493
C000             TXTTAB   =    43        ; TXTTAB = 45 on the 128—start-of-BASIC
                                         ; pointer
C000             VARTAB   =    45        ; TEXTTP = 4624 on the 128—end-of-BASIC
                                         ; pointer
C000             LINKPG   =    42291     ; LINKPG = 20303 on the 128
C000             SETBNK   =    65384     ; Kernal bank number for load and filename
                                         ; (128 only)
C000             MMUREG   =    65280     ; MMU configuration register (128 only)
                                         ;
                                         ; Load the directory into normal BASIC
                                         ; memory
                                         ;
                                         ; Open channel 15 here if you include disk
                                         ; error checking (DERRCK).
                                         ;
C000             DIRPRG   =    *
                                         ; LDA #0; set for bank 15 (128 only)
                                         ; STA MMUREG, (128 only)
C000  A9 01               LDA  #1        ; logical file number (value doesn't matter)
C002  A2 08               LDX  #8        ; device number for disk drive
C004  A0 00               LDY  #0        ; secondary address of zero causes relative
                                         ; load
C006  20 BA FF            JSR  SETLFS    ; set parameters for relative load
                                         ; Include the following three instructions
                                         ; for the 128 only.
                                         ; LDA BNKNUM; bank containing the
                                         ; program
                                         ; LDX BNKFNM; bank containing the
                                         ; ASCII filename
                                         ; JSR SETBNK
C009  A9 02               LDA  #FNLENG   ; length of filename
C00B  A2 22               LDX  #<FILENM  ; the filename is "$0"
C00D  A0 C0               LDY  #>FILENM
C00F  20 BD FF            JSR  SETNAM    ; set up filename
```

```
C012  A9 00               LDA   #0          ; flag for load
C014  A6 2B               LDX   TXTTAB      ; low byte of start-of-BASIC program
                                            ; address
C016  A4 2C               LDY   TXTTAB+1    ; high byte of start-of-BASIC program
                                            ; address
C018  20 D5 FF            JSR   LOAD        ; load the directory at the start of BASIC
                                            ;
                                            ; JSR DERRCK; Insert here for disk error
                                            ; checking.
                                            ;
                                            ; Change VARTAB in the next two
                                            ; instructions to TEXTTP on the 128.
C01B  86 2D               STX   VARTAB      ; store end of directory address into end-of-
                                            ; BASIC program pointer
C01D  84 2E               STY   VARTAB+1
C01F  4C 33 A5            JMP   LINKPG      ; relink lines of tokenized program text and
                                            ; RTS
                                            ;
                                            ; Insert DERRCK here if you're including
                                            ; disk error checking.
                                            ;
C022  24 30      FILENM   .ASC  "$0"        ; directory name
C024             FNLENG   =     * - FILENM  ; length of filename
                                            ; Include the next two variables for the 128
                                            ; only.
                                            ; BNKNUM .BYTE 0; bank number where
                                            ; program is to be loaded
                                            ; BNKFNM .BYTE 0; bank number where
                                            ; ASCII filename is located
```

See also DIRBYT, FRESEC.

Name

Disable RUN/STOP–RESTORE

Description

DISRSR disables the reset function of the RUN/STOP–RESTORE key combination by redirecting the NMI interrupt vector to the end of the normal NMI interrupt handler.

Prototype

Change the NMI interrupt vector to point to a harmless routine that skips the normal interrupt handling.

Explanation

There are two normal sources for an NMI interrupt in the 64 and 128. One is the CIA (Complex Interface Adapter) #2 chip, which generates the interrupts to handle RS-232 communications. The other is the RESTORE key.

DISRSR changes the NMI interrupt vector so that it skips both sources of NMI interrupts. Note that, in addition to disabling RUN/STOP–RESTORE, this technique will also disable RS-232 communications through the user port.

On the 64, this is accomplished by pointing the NMI vector directly to the RTI instruction at the end of the normal NMI service routine. The 128 pushes the A, X, and Y registers, as well as the configuration register, onto the stack just before jumping through the NMI vector. As a result, before leaving the routine, you have to restore these registers. This is done by jumping to the common IRQ exit routine at 65331.

On the 64, the A, X, and Y registers are also stored on the stack, but as part of the NMI interrupt handler routine itself. Since we skip these instructions altogether on this machine, you don't need to restore the registers before exiting the routine.

Routine

```
C000                NMIVEC   =     792          ; vector to nonmaskable interrupt routine
C000                RTINMI   =     65217        ; RTINMI = 65331 on the 128—return from
                                                ; NMI routine address
                                                ;
                                                ; Disable RUN/STOP-RESTORE key
                                                ; sequence by skipping NMI handler.
C000  A9 C1         DISRSR   LDA   #<RTINMI     ; redirect NMI vector, low byte first
C002  8D 18 03               STA   NMIVEC
C005  A9 FE                  LDA   #>RTINMI     ; then high byte
C007  8D 19 03               STA   NMIVEC+1
C00A  60                     RTS                ; we're done
```

See also DISTOP, ERRRDT, RSTVEC.

Name

Disable the STOP key by changing the STOP vector

Description

DISTOP disables the STOP key by redirecting the STOP vector past the STOP key check in the normal STOP handler.

Prototype

Store the address of that portion of the STOP routine that is just beyond the STOP key check into the STOP vector and RTS.

Explanation

The STOP vector at location 808 is one of Kernal indirect vectors in page 3. This vector ordinarily points to a short ROM routine that checks whether the STOP key is pressed.

Press the STOP key, and a $7F is stored into the STOP key flag at location $91. The Kernal STOP routine, when called, determines whether the STOP key flag contains this value. This routine begins with the same series of instructions on the 128 as on the 64. The only difference in the two is the address of the routine—on the 128, it's at 63086.

On the 64, the code for this routine goes like this:

```
F6ED  A5 91     LDA $91
F6EF  C9 7F     CMP #$7F
F6F1  D0 07     BNE $F6FA
                .
                .
                .
F6FA  60        RTS
```

In **DISTOP**, we disable the STOP key by pointing the STOP vector to the CMP at $F6EF. Consequently, since the accumulator never gets the $7F from location $91, the routine always branches to the RTS at $F6FA.

Routine

```
C000                 STOPVC   =    808          ; vector to Kernal STOP key routine
C000                 STOP     =    63213        ; STOP = 63086 on the 128—STOP routine
                                                ; address
                                                ;
                                                ; Disable STOP key by skipping STKEY flag
                                                ; check.
C000  A9 EF          DISTOP   LDA  #<STOP+2     ; redirect STOP vector ahead by two bytes
C002  8D 28  03               STA  STOPVC       ; change low byte first
C005  A9 F6                   LDA  #>STOP+2     ; and then high byte
C007  8D 29  03               STA  STOPVC+1
C00A  60                      RTS               ; we're done
```

See also DISRSR, ERRRDT, RSTVEC.

Name

Divide one byte value by another and store the result (and remainder) in memory

Description

This version of the division routine repeatedly subtracts the second number from the first. The leftover number is kept in REMAIN. The result is in TOTAL.

Prototype

1. Store the first number in FIRST and the second in SECOND.
2. Zero out the total and remainder.
3. Load the accumulator from FIRST.
4. Compare to SECOND.
5. If the carry flag is clear, store the remainder in REMAIN and exit.
6. INC the total and subtract SECOND from FIRST.
7. Branch back to step 4.

Explanation

When you're dealing with byte-sized quantities (0–255), dividing by repeated subtraction of one number from another will suffice. To divide 99 by 10, just subtract 10 until you have a number smaller than 10. Whatever is left is the remainder.

For division of larger numbers, see **DIVINT**.

Routine

```
C000                LINPRT   =    $BDCD     ; LINPRT = $8E32 on the 128
C000                CHROUT   =    $FFD2
                                            ;
C000  20  19  C0             JSR  DIVBYT    ; divide them
C003  A9  00                 LDA  #0
C005  AE  39  C0             LDX  TOTAL     ; print the result
C008  20  CD  BD             JSR  LINPRT
C00B  A9  0D                 LDA  #13       ; print RETURN
C00D  20  D2  FF             JSR  CHROUT
C010  A9  00                 LDA  #0
C012  AE  3A  C0             LDX  REMAIN    , print the remainder
C015  20  CD  BD             JSR  LINPRT
C018  60                     RTS
                                            ;
C019  A9  00        DIVBYT   LDA  #0        ; zero out the total
C01B  8D  39  C0             STA  TOTAL     ; store it in TOTAL
C01E  8D  3A  C0             STA  REMAIN    ; and remainder
C021  AD  37  C0             LDA  FIRST     ; get the number
C024  CD  38  C0    VLOOP    CMP  SECOND    ; compare it with the second
C027  90  0A                 BCC  DONE      ; SECOND is bigger
C029  EE  39  C0             INC  TOTAL     ; else, increment the result
C02C  F0  08                 BEQ  NOREM     ; no remainder
C02E  ED  38  C0             SBC  SECOND    ; carry is set, so subtract
```

```
C031  B0 F1             BCS   VLOOP     ; branch always (carry is set)
                                        ;
C033  8D 3A C0  DONE    STA   REMAIN    ; .A holds the remainder
C036  60        NOREM   RTS             ; end the subroutine
                                        ;
C037  64        FIRST   .BYTE 100
C038  03        SECOND  .BYTE 3
C039  00        TOTAL   .BYTE 0
C03A  00        REMAIN  .BYTE 0
```

See also DIVFP, DIVINT.

Name

Divide one floating-point number by another

Description

Like most of the other floating-point routines in this book, **DIVFP** depends on built-in BASIC routines. The example program divides 30,000 by 302 and prints out the result, complete with decimal fractions.

Prototype

1. Set up the dividend (or numerator) in floating-point accumulator 2 (FAC2).
2. Put the divisor (or denominator) in FAC1.
3. Call the FDIVT routine in ROM. The answer can be found in FAC1.

Explanation

The framing program converts the integer value 30,000 to a floating-point number with GIVAYF. The MOVEF routine moves it from FAC1 to FAC2. Next, the number 302 is stored into FAC1, and the **DIVFP** routine is called (a simple ROM call). Finally, FOUT converts the contents of FAC1 to ASCII numbers, which are then printed to the screen.

Routine

```
C000                ZP       =    $FB
C000                CHROUT   =    $FFD2
C000                FDIVT    =    $BB12      ; FDIVT = $8B4C on the 128—divide FAC2
                                             ; by FAC1; result in FAC1
C000                MOVEF    =    $BC0F      ; MOVEF = $8C3B on the 128—moves FAC1
                                             ; to FAC2
C000                GIVAYF   =    $B391      ; GIVAYF = $AF03 on the 128—converts
                                             ; integer to floating point
C000                FOUT     =    $BDDD      ; FOUT = $8E42 on the 128—converts FAC1
                                             ; to ASCII string
                                             ;
                                             ; Convert the numbers 30000 and 302 to
                                             ; floating point and divide.
C000  A9 75                   LDA  #>30000   ; high byte of 30000
C002  A0 30                   LDY  #<30000   ; low byte
C004  20 91 B3                JSR  GIVAYF    ; convert it; now it's in FAC1
C007  20 0F BC                JSR  MOVEF     ; move FAC1 to FAC2
C00A  A9 01                   LDA  #>302     ; high byte of 302
C00C  A0 2E                   LDY  #<302     ; low byte
C00E  20 91 B3                JSR  GIVAYF    ; convert it
                                             ; FAC1 now holds 302; FAC2 holds 30000.
C011  20 29 C0                JSR  DIVFP     ; divide 30000 by 302; the result is in FAC1
C014  20 DD BD                JSR  FOUT      ; convert to ASCII
C017  85 FB                   STA  ZP        ; pointer
C019  84 FC                   STY  ZP+1      ; to the string
```

```
C01B  A0  00            LDY   #0
C01D  B1  FB    PRTLOP  LDA   (ZP),Y
C01F  D0  01            BNE   PRNIT
C021  60                RTS
C022  20  D2 FF PRNIT   JSR   CHROUT
C025  C8                INY
C026  D0  F5            BNE   PRTLOP
C028  60                RTS
                                      ;
C029  20  12 BB DIVFP   JSR   FDIVT   ; divide FAC2 by FAC1
C02C  60                RTS           ; the result is in FAC1
```

See also DIVBYT, DIVINT.

DIVINT

Name

Divide one integer value into another

Description

For values that take up two bytes or more, this division routine is preferable to the subtraction method used in **DIVBYT**. It's much faster than subtracting.

Prototype

1. Since there are 16 bits in a two-byte integer, store a 16 into a counter (change this if you're using larger numbers).
2. Store zeros into ANSWER and WORK, which will eventually contain the answer and the remainder.
3. Copy the numerator, also called the dividend, from DIVNUM to a work area COPYN.
4. Begin division: Rotate COPYN to the left. The additional bit rotates into WORK.
5. Compare the contents of WORK to DIVDEN (the denominator or divisor).
6. If WORK is equal or larger, the carry flag will be set. Rotate the set carry (a 1) left into ANSWER and execute step 7.
7. Subtract DIVDEN from WORK and store the result in WORK. Skip step 8.
8. If, after step 5, WORK was smaller, carry would be clear. Rotate this zero bit left into ANSWER.
9. Decrement the counter setup in step 1. If it's not yet zero, loop back to step 4.

Explanation

The following partial example of a binary division may be helpful in understanding how division works in ML:

```
       0001
110 ) 10110010
        110
        ---
        110
```

The 110 is the denominator (or divisor) being divided into 10110010, the numerator (or dividend). There's a third work area, called WORK in the program below, which starts out

holding a zero. The main loop rotates DIVDEN (10110010 in the example above) to the left, and the high bit goes into WORK:

```
   WORK      DIVDEN
1  00000001  0110010x
2  00000010  110010xx
3  00000101  10010xxx
4  00001011  0010xxxx
```

As you can see, the number in WORK gradually grows larger as more bits are shifted left (the *x*'s represent unknown bits that don't matter). Since the example is dividing by the number 110, at each step, we have to compare WORK to the denominator. The binary numbers %1, %10, and %101 are smaller, so the carry flag is clear, and a zero gets rotated into ANSWER. Note the first three zeros in the example.

When WORK is equal to or larger than DIVDEN, carry is set (which means a 1 gets rotated into the answer), and we have to subtract DIVDEN from WORK. Then the rotate instructions and compares continue.

After division is complete, the answer is held in ANSWER. The remainder can be found in WORK. The example program divides two numbers (3112/550) and prints the answer. The remainder, preceded by the letter *R* is also printed.

To use this routine in your own programs, store the integer values in DIVNUM and DIVNUM. Using the bit-shifting method is faster than subtracting. Dividing 60,000 by 3, for example, would require 30,000 loops in **DIVBYT**, but only 16 in **DIVINT**.

Note: If you're dividing by a power of 2 (2, 4, 8, 16, 32, and so forth), you can skip this routine and simply shift the dividend to the right, with LSR for the high byte and ROR for any intermediate or lower bytes.

Warning: Division by zero is mathematically illegal, and this program doesn't contain a trap for zero. If you think a user might try dividing by zero, you'll need to check for zeros at the beginning of **DIVINT**.

Routine

```
C000              LINPRT   =    $BDCD       ; LINPRT = $8E32 for the 128
C000              CHROUT   =    $FFD2
                                            ;
C000  A9 93                LDA  #147        ; clear screen
C002  20 D2 FF             JSR  CHROUT      ; print it
C005  AE 41 C0             LDX  DIVNUM      ; low byte of the numerator or dividend
C008  AD 42 C0             LDA  DIVNUM+1    ; high byte
```

```
C00B  20 CD BD              JSR   LINPRT        ; print it
C00E  A9 2F                 LDA   #47           ; the slash (/), to indicate division
C010  20 D2 FF              JSR   CHROUT        ; print it
C013  AE 43 C0              LDX   DIVDEN        ; low byte of the denominator or divisor
C016  AD 44 C0              LDA   DIVDEN +1     ; high byte
C019  20 CD BD              JSR   LINPRT        ; print it
C01C  A9 0D                 LDA   #13           ; print RETURN
C01E  20 D2 FF              JSR   CHROUT        ; new line
C021  20 4C C0              JSR   DIVINT        ; divide the numbers
C024  AE 47 C0              LDX   ANSWER        ; and print the answer
C027  AD 48 C0              LDA   ANSWER + 1
C02A  20 CD BD              JSR   LINPRT
C02D  A9 0D                 LDA   #13           ; print RETURN again
C02F  20 D2 FF              JSR   CHROUT
C032  A9 52                 LDA   #82           ; letter R for remainder
C034  20 D2 FF              JSR   CHROUT        ; print it, then
C037  AE 45 C0              LDX   REMAIN        ; low byte of remainder
C03A  AD 46 C0              LDA   REMAIN +1     ; high byte
C03D  20 CD BD              JSR   LINPRT        ; print it
C040  60                    RTS                 ; and quit
                                                ;
C041  28 0C       DIVNUM    .WORD 3112          ; 3112 will be divided by
C043  26 02       DIVDEN    .WORD 550           ; 550
C045  00 00       WORK      .BYTE 0,0
C047              REMAIN    =     WORK          ; the remainder will end up in WORK (also
                                                ; known as REMAIN)
C047  00 00       ANSWER    .BYTE 0,0
C049  00 00       COPYN     .BYTE 0,0
C04B  00          COUNTR    .BYTE 0
                                                ;
C04C  20 61 C0    DIVINT    JSR   SETUP         ; set the counter to 16
C04F  20 67 C0              JSR   ZEROS         ; zero out WORK and ANSWER
C052  20 72 C0              JSR   COPYNM        ; copy DIVNUM to COPYN
C055  20 7F C0    DIVLP     JSR   MVOVER        ; rotate COPYN and WORK to the left
C058  20 8C C0              JSR   DIVIDE        ; the main division routine
C05B  CE 4B C0              DEC   COUNTR        ; count down
C05E  D0 F5                 BNE   DIVLP         ; if it's not zero yet, keep going
C060  60                    RTS                 ; quit the DIVINT routine
                                                ;
                                                ; Setup just puts a 16 into COUNTR.
                                                ; 16 represents the number of bits in
                                                ; DIVNUM.
C061  A9 10       SETUP     LDA   #16
C063  8D 4B C0              STA   COUNTR
C066  60                    RTS
                                                ;
                                                ; Next, copy zeros into WORK and
                                                ; ANSWER.
C067  A9 00       ZEROS     LDA   #0
C069  A0 03                 LDY   #3
C06B  99 45 C0    ZLOOP     STA   WORK,Y
C06E  88                    DEY
C06F  10 FA                 BPL   ZLOOP         ; as long as .Y is zero or higher, loop back
C071  60                    RTS
                                                ;
                                                ; Copy DIVNUM to COPYN.
C072  AD 41 C0    COPYNM    LDA   DIVNUM
C075  8D 49 C0              STA   COPYN
C078  AD 42 C0              LDA   DIVNUM+1
C07B  8D 4A C0              STA   COPYN+1
C07E  60                    RTS
                                                ;
                                                ; Move a bit to the left from COPYN to
                                                ; WORK.
```

```
C07F  0E 49 C0  MVOVER  ASL  COPYN       ; low-byte shifts left
C082  2E 4A C0          ROL  COPYN+1     ; into high byte
C085  2E 45 C0          ROL  WORK        ; into WORK
C088  2E 46 C0          ROL  WORK+1      ; and high byte of WORK
C08B  60                RTS
                                         ;
C08C  AD 46 C0  DIVIDE  LDA  WORK+1      ; high byte of WORK
C08F  CD 44 C0          CMP  DIVDEN+1    ; compare to the divisor
C092  F0 09             BEQ  LOOKMR      ; look more (check the low byte) if equal
C094  B0 0F             BCS  SUBTR       ; WORK is higher, so subtract
                                         ; If we fall through from above, carry is
                                         ; clear.
C096  2E 47 C0  FIXANS  ROL  ANSWER      ; move the carry flag into ANSWER
C099  2E 48 C0          ROL  ANSWER+1    ; high byte, too
C09C  60                RTS              ; end of FIXANS and/or subroutine
                                         ;
C09D            LOOKMR  =    *           ; check the low byte if the high byte was
                                         ; equal
C09D  AD 45 C0          LDA  WORK        ; get value in WORK
C0A0  CD 43 C0          CMP  DIVDEN      ; compare to denominator (divisor) low
                                         ; byte
C0A3  90 F1             BCC  FIXANS      ; if carry is clear, DIVDEN is bigger, so exit
                                         ;
C0A5            SUBTR   =    *           ; else subtract DIVDEN from WORK
C0A5  20 96 C0          JSR  FIXANS      ; carry is always set (note the RTS of
                                         ; FIXANS returns to here)
C0A8  38                SEC              ; carry was changed by FIXANS, so set it
C0A9  AD 45 C0          LDA  WORK
C0AC  ED 43 C0          SBC  DIVDEN
C0AF  8D 45 C0          STA  WORK        ; subtract DIVDEN from WORK
C0B2  AD 46 C0          LDA  WORK+1      ; high byte, too
C0B5  ED 44 C0          SBC  DIVDEN+1
C0B8  8D 46 C0          STA  WORK+1
C0BB  60                RTS
```

See also DIVBYT, DIVFP.

Name

Change the ERROR vector

Description

ERRRDT redirects BASIC's ERROR vector to your own routine.

Prototype

Store the address of the custom error routine into the ERROR vector; then RTS.

Explanation

When an error occurs during a BASIC program, an indirect jump is taken through the ERROR vector at location 768. This vector normally points to the ROM routine which displays the appropriate one of the familiar BASIC error messages, such as SYNTAX ERROR, ILLEGAL QUANTITY ERROR, and so forth. In some cases, however, you may want to substitute a custom error message in place of the standard one. In this case, you can change the address in the ERROR vector to point to an error message routine of your own.

For example, when you type in BASIC programs that contain many numeric DATA statements being POKEd into memory, you'll frequently get an error that's difficult to pin down. If you accidentally include a number higher than 255 and run the program, you'll get the error message ?ILLEGAL QUANTITY IN LINE *xxx*. But the line given as *xxx* is the one containing the READ statement rather than the one with the errant data. The READ works just fine (it's legal to READ numbers greater than 255), but the POKE causes the problem.

The example program relies on **ERRRDT** to solve this problem. Ordinarily, the ERROR vector points to a routine that prints either a BASIC error message or the READY prompt. Using the .X register, this routine locates the error message in a table and then prints it. If you're in program mode, the number of the line that's currently being executed is taken from CURLIN (location 57 on the 64; 59 on the 128) and is printed as well.

ERRRDT changes the ERROR vector to point to our own custom error handler at EWEDGE. If an error other than an illegal quantity error occurs (.X <> 14), normal error handling will result. But if .X contains a 14 upon entry into EWEDGE—meaning an illegal quantity has occurred—the current DATA line number (CURLIN) will be stored into the current BASIC

line (DATLIN) before the normal error handler will execute. And so, in our example above, instead of telling us that the error occurred in the line with the READ statement, with this routine in place, BASIC reports the actual DATA line containing the typo.

Of course, this routine fails to distinguish among the many possible sources of illegal quantity errors. If your program contains a POKE 251,257, for instance, the error message that results will erroneously point you to the last DATA line that was read. Because of this, you should limit the use of this wedge to BASIC programs that contain many numeric DATA statements—primarily BASIC loaders of ML object code.

Routine

```
C000                ERRVEC   =     768          ; error vector
C000                ERRNOR   =     58251        ; ERRNOR = 19775 on the 128—normal
                                                ; error-service routine
C000                CURLIN   =     57           ; CURLIN = 59 on the 128—current BASIC
                                                ; line being executed
C000                DATLIN   =     63           ; DATLIN = 65 on the 128—current data
                                                ; line
                                                ;
                                                ; Insert a custom error routine that looks for
                                                ; an illegal quantity error.
                                                ; Assume it occurs while reading data and
                                                ; report the data line number.
                                                ;
                                                ; ERRRDT points the ERRVEC vector to our
                                                ; routine.
C000 A9 0B          ERRRDT   LDA   #<EWEDGE     ; low byte first
C002 8D 00 03                STA   ERRVEC
C005 A9 C0                   LDA   #>EWEDGE     ; then high byte
C007 8D 01 03                STA   ERRVEC+1
C00A 60                      RTS                ; and exit the setup routine
                                                ;
                                                ; Upon entry, .X contains the error number
                                                ; We let the system handle
                                                ; all errors except the illegal quantity error
                                                ; (error 14).
C00B E0 0E          EWEDGE   CPX   #14          ; is it an illegal quantity error?
C00D D0 08                   BNE   EXIT         ; if not, exit through the normal error handler
                                                ; Otherwise, substitute the current data line
                                                ; for the current BASIC line.
C00F A5 3F                   LDA   DATLIN       ; low byte first
C011 85 39                   STA   CURLIN
C013 A5 40                   LDA   DATLIN+1     ; then high byte
C015 85 3A                   STA   CURLIN+1
C017 4C 8B E3       EXIT     JMP   ERRNOR       ; and execute the normal error handler
                                                ; routine
```

See also DISRSR, DISTOP, RSTVEC.

Name

Produce an explosion sound

Description

EXPLOD provides the sound of an explosion and could be used in any number of game programs, with or without modification.

Prototype

1. Clear the SID chip with **SIDCLR**.
2. Set the necessary SID chip parameters (volume, attack/decay, sustain/release, and frequency).
3. Select the noise waveform and gate the sound.
4. Cause a delay (here, 120 jiffies), and then start the release cycle (ungate the sound).
5. Then RTS.

Explanation

This routine relies on the noise waveform to achieve its effect. You can alter the sound that's produced by varying a number of parameters in the routine. These include the attack/decay and sustain/release rates, the base frequency for the noise waveform, and the number of jiffies between gating and ungating the chip.

EXPLOD is no different in one respect from other sound-effect routines in this book. After the release cycle is complete, the SID chip hums on in the background. Again, to prevent this, after the explosion has sounded, store zeros in the frequency registers (FREHI1, FREHI3) or turn the chip off altogether by JSRing to **SIDCLR**.

Routine

```
C000                SIGVOL   =    54296     ; SID chip volume register
C000                ATDCY1   =    54277     ; voice 1 attack/decay register
C000                SUREL1   =    54278     ; voice 1 sustain/release register
C000                FRELO1   =    54272     ; voice 1 frequency control (low byte)
C000                FREHI1   =    54273     ; voice 1 frequency control (high byte)
C000                VCREG1   =    54276     ; voice 1 control register
C000                JIFELO   =    162       ; low byte of jiffy clock
                                            ;
C000  20 2F C0      EXPLOD   JSR  SIDCLR    ; clear the SID chip
C003  A9 0F                  LDA  #15       ; set volume
C005  8D 18 D4               STA  SIGVOL
C008  A9 0C                  LDA  #$0C      ; set attack/decay
C00A  8D 05 D4               STA  ATDCY1
C00D  A9 18                  LDA  #$18      ; set sustain/release
C00F  8D 06 D4               STA  SUREL1
C012  A9 00                  LDA  #0        ; set voice 1 low frequency
C014  8D 00 D4               STA  FRELO1
C017  A9 18                  LDA  #24       ; set voice 1 high frequency
```

```
C019  8D 01  D4            STA   FREHI1
C01C  A9 81                LDA   #%10000001    ; select noise waveform and gate sound
C01E  8D 04  D4            STA   VCREG1
C021  A9 78                LDA   #120          ; cause a delay of 120 jiffies
C023  65 A2                ADC   JIFFLO        ; add current jiffy reading
C025  C5 A2      DELAY     CMP   JIFFLO        ; and wait for 120 jiffies to elapse
C027  D0 FC                BNE   DELAY
C029  A9 80                LDA   #%10000000    ; ungate sound
C02B  8D 04  D4            STA   VCREG1
C02E  60                   RTS
                                               ;
                                               ; Clear the SID chip.
C02F  A9 00      SIDCLR    LDA   #0            ; fill with zeros
C031  A0 18                LDY   #24           ; index to FRELO1
C033  99 00  D4  SIDLOP    STA   FRELO1,Y      ; store zero in SID chip address
C036  88                   DEY                 ; for next lower byte
C037  10 FA                BPL   SIDLOP        ; fill 25 bytes
C039  60                   RTS
```

See also BEEPER, BELLRG, INTMUS, MELODY, NOTETB, SIDCLR, SIDVOL, SIRENS.

Name

Print floating-point accumulator 1 to a specified number of decimal places

Description

If you print a floating-point variable, anywhere from zero to nine decimal places may be displayed. In many situations, you'll want to format your numeric output. With **FACPRD**, you can do just that. This routine lets you specify the number of decimal places to print when you're outputting floating-point numbers to the screen. In the process, no rounding occurs.

Prototype

1. Enter this routine with the number of decimal places to print in DECIML.
2. Keep a counter of digits past the decimal in zero page.
3. Load each character from the number string.
4. If the end of the string is reached (a zero byte occurs), print a decimal point and/or the proper number of trailing zeros (in OUTCHK).
5. Increase the decimal counter if the decimal point has been printed.
6. Otherwise, check the current character for a decimal point. If one occurs, increase the decimal counter.
7. Check to see whether zero decimal places have been requested. If so, exit the routine.
8. Determine whether the last decimal place has been printed. If so, place a terminator byte of zero at the end of the number string.
9. Print the current character and branch back to step 3.

Explanation

This program is much like the example program shown under **FACPRT**, where a floating-point number—365.25—is converted to an ASCII string and printed to the screen. Again, in this routine, the number 365.25 is printed. Here, however, you have the option of specifying the number of decimal places (0–9) that are displayed. Notice that **CHRGTR** allows only numeric input, with the exception of the RETURN key. Pressing RETURN exits the program.

FACPRD takes the ASCII string in the workspace area at the top of the stack (beginning at $100) and displays it to the number of decimal places in DECIML. The routine begins by

initializing a decimal-place counter in zero page to $FF. Each character from the string is then examined to see whether it's a terminator byte (zero) or a decimal point.

If a terminator byte occurs, we branch to the routine OUTCHK. OUTCHK prints a decimal point (if needed) and the proper number of trailing zeros.

If a decimal point occurs, increment the decimal counter and print the decimal point if one or more decimal places have been requested. As a result, the counter will contain a positive value once the decimal point has been printed. On the other hand, if DECIML is zero (no decimal places have been specified), we simply exit the routine.

Assuming the decimal point has been printed, before we print each character from the string, the decimal counter is compared to DECIML (the number of decimal places requested). If they agree in value, a terminator byte is placed at the next character position within the string. So, after the current character is printed, the next character (the zero byte) will send us to OUTCHK where trailing zeros can be added if necessary.

Routine

```
C000                ZP      =     251
C000                CHROUT  =     65490
C000                GETIN   =     65508
C000                FAC1    =     97        ; FAC1 = 99 on the 128—floating-point
                                            ; accumulator 1
C000                FOUT    =     48605     ; FOUT = 36418 on the 128—converts FAC1
                                            ; to ASCII
C000                STWORK  =     256       ; workspace at the top of the stack
                                            ;
                                            ; Print the number in floating-point
                                            ; accumulator 1 to the number
                                            ; of decimal places requested. Quit on
                                            ; RETURN.
C000  A9 93         CLRCHR  LDA   #147      ; clear the screen
C002  20 D2 FF              JSR   CHROUT
C005  A0 00         OUTLOP  LDY   #0        ; as an index for PRTLOP
C007  B9 8C C0      PRTLOP  LDA   STRING,Y  ; print the prompt "NUMBER OF DECIMAL
                                            ; PLACES (0–9)?"
C00A  F0 06                 BEQ   CHRGTR    ; if zero byte, skip ahead
C00C  20 D2 FF              JSR   CHROUT    ; print each character in the prompt
C00F  C8                    INY             ; next character
C010  D0 F5                 BNE   PRTLOP    ; branch always
C012  20 E4 FF      CHRGTR  JSR   GETIN     ; get a keypress in the range 0–9, or a
                                            ; RETURN
C015  F0 FB                 BEQ   CHRGTR    ; if no keypress
C017  C9 0D                 CMP   #13       ; is it RETURN?
C019  F0 27                 BEQ   EXIT      ; if so, then quit
C01B  CD AD C0              CMP   RANGE1    ; is it zero?
C01E  90 F2                 BCC   CHRGTR    ; if it's less, get another key
C020  CD AE C0              CMP   RANGE2    ; is it 9 plus 1?
C023  B0 ED                 BCS   CHRGTR    ; if it's more, get another key
C025  29 0F                 AND   #15       ; put ASCII number in a range 0–9
C027  8D B5 C0              STA   DECIML    ; store .A for FACPRD
```

```
C02A  A0 05           LDY  #5            ; index to floating-point number
C02C  B9 AF C0  LOOP  LDA  FPNUM,Y       ; store each byte of FPNUM in FAC1
C02F  99 61 00        STA  FAC1,Y
C032  88              DEY                ; for next byte
C033  10 F7           BPL  LOOP          ; if .Y is 0-5, continue
C035  20 DD BD        JSR  FOUT          ; convert contents of FAC1 to ASCII string
                                         ; string is in stack area
C038  20 43 C0        JSR  FACPRD        ; print the FAC1 to DECIML decimal places
C03B  A9 0D           LDA  #13           ; print RETURN
C03D  20 D2 FF        JSR  CHROUT
C040  D0 C3           BNE  OUTLOP        ; handle another request
C042  60        EXIT  RTS
                                         ;
                                         ; FACPRD displays the number in FAC1 to a
                                         ; number (DECIML) of decimal places.
C043  A0 00     FACPRD LDY #0            ; as an index
C045  A2 FF           LDX  #255          ; as a decimal counter
C047  86 FB           STX  ZP            ; store decimal counter in zero page
C049  B9 00 01  MORE  LDA  STWORK,Y      ; load each ASCII byte of string
C04C  F0 20           BEQ  OUTCHK        ; if zero byte, print decimal and/or trailing
                                         ; zeros
C04E  A6 FB           LDX  ZP            ; check decimal counter
C050  10 04           BPL  INCRZP        ; increase decimal counter if decimal has
                                         ; already been reached
C052  C9 2E           CMP  #46           ; is it currently a decimal point?
C054  D0 12           BNE  PRINT         ; no, so print .A
C056  E6 FB     INCRZP INC ZP            ; increment decimal counter
C058  AE B5 C0        LDX  DECIML        ; load with number of decimal places
                                         ; requested
C05B  F0 2E           BEQ  OUT           ; if zero decimal points requested
C05D  E4 FB           CPX  ZP            ; compare with decimal-place counter
C05F  D0 07           BNE  PRINT         ; we haven't reached the last one, so print
                                         ; .A
C061  48              PHA                ; save .A
C062  A9 00           LDA  #0            ; put terminator character in the position
                                         ; which follows
C064  99 01 01        STA  STWORK+1,Y
C067  68              PLA                ; restore .A
C068  20 D2 FF  PRINT JSR  CHROUT        ; print a character
C06B  C8              INY                ; next character
C06C  D0 DB           BNE  MORE          ; branch always
C06E  AE B5 C0  OUTCHK LDX DECIML        ; see whether decimal and/or extra zeros
                                         ; need printing
C071  E4 FB           CPX  ZP            ; have all decimal places been printed?
C073  F0 16           BEQ  OUT           ; yes, so get out
C075  B0 05           BCS  DECIZR        ; if carry set, we need to print one or more
                                         ; trailing zeros
C077  A9 2E           LDA  #46           ; otherwise, print a decimal point
C079  20 D2 FF        JSR  CHROUT
C07C  AD B5 C0  DECIZR LDA DECIML        ; subtract decimal counter from requested
                                         ; number of places
C07F  38              SEC
C080  E5 FB           SBC  ZP
C082  AA              TAX                ; we'll fill remainder with zeros
C083  A9 30           LDA  #48
C085  20 D2 FF  ZRLOOP JSR CHROUT        ; print a zero
C088  CA              DEX
C089  D0 FA           BNE  ZRLOOP        ; if more to print, continue
C08B  60              RTS
                                         ;
C08C  4E 55 4D  STRING .ASC "NUMBER OF DECIMAL PLACES (0-9)?"
```

```
C0AB  0D  00                 .BYTE  13,0               ; carriage return and terminator byte
                                                       ;
C0AD  30          RANGE1     .BYTE  48                 ; ASCII 0
C0AE  3A          RANGE2     .BYTE  58                 ; ASCII 9 plus 1
C0AF  89  B6  A0  FPNUM      .BYTE  137,182,160,0,0,0
                                                       ; the value for 365.25 in FP accumulator
C0B5  00          DECIML     .BYTE  0                  ; storage for number of decimal places
```

See also BYTASC, CNUMOT, FACPRT, NUMOUT.

Name

Print the value in floating-point accumulator 1

Description

All BASIC mathematical operations use a series of six locations—known collectively as a *floating-point accumulator*—to store real numbers. Actually, the 64 and 128 have two separate floating-point accumulators. The primary one, located at 97–102 on the 64 and 99–104 on the 128, is labeled FAC1. The secondary one, often used to hold an interim value in a calculation, is FAC2 (located at 105–110 on the 64 and 107–112 on the 128).

At any rate, whether you use BASIC's built-in routines as they are, modify them, or write your own, you'll certainly need to display the contents of these floating-point accumulators at some point. The routine that follows prints the contents of floating-point accumulator 1 to the screen.

Prototype

1. Prior to the routine, JSR to FOUT to convert the contents of floating-point accumulator 1 to an ASCII string at $100.
2. Beginning at $100, print each byte of the string until a zero byte is found.

Explanation

In the example program, the number 365.25—the number of days in a year—is represented by FPNUM, just as it would appear in one of the floating-point accumulators. The first byte of FPNUM is the binary exponent of the number (plus 129 to account for negative exponents)—that is $137 - 129$, which is 8, so the exponent is 2 to the eighth power. The next four bytes are the mantissa of the number, with the first bit in the series containing the sign of the number. The last byte is the sign byte—0 indicates a positive number; 255, a negative number.

In the program, the floating-point representation of 365.25 is stored in floating-point accumulator 1. The BASIC routine FOUT (located at 48605 on the 64 and 36418 on the 128) converts it into an ASCII string and stores it in a workspace area at the top of the stack (beginning at $100). After the number has been converted, **FACPRT** prints it to the screen.

In converting the floating-point number to an ASCII string, FOUT positions a terminator byte of zero at the end of

the string. As a result, this routine is much like other string-printing routines in this book. Using CHROUT, you simply output each byte of the string to the screen until a zero byte is reached.

Routine

```
C000                 CHROUT   =     65490
C000                 FAC1     =     97            ; FAC1 = 99 on the 128—floating point
                                                  ; accumulator 1
C000                 FOUT     =     48605         ; FOUT = 36418 on the 128—converts FAC1
                                                  ; to ASCII
C000                 STWORK   =     256           ; workspace at the top of the stack
                                                  ;
                                                  ; Print the number in floating-point
                                                  ; accumulator 1.
C000                 MAIN     =     *
C000  A9 93          CLRCHR   LDA   #147          ; clear the screen
C002  20 D2 FF                JSR   CHROUT
C005  A0 05                   LDY   #5            ; index to floating-point number
C007  B9 24 C0       LOOP     LDA   FPNUM,Y       ; store each byte of FPNUM in FAC1
C00A  99 61 00                STA   FAC1,Y
C00D  88                      DEY
C00E  10 F7                   BPL   LOOP          ; if .Y is 0–5, continue
C010  20 DD BD                JSR   FOUT          ; convert contents of FAC1 to ASCII string
                                                  ; string is in stack area
C013  4C 16 C0                JMP   FACPRT        ; print the FAC1 value and return
                                                  ;
                                                  ; FACPRT prints the number in floating-
                                                  ; point accumulator 1.
C016  A0 [redacted]  FACPRT   LDY   #0            ; as an index
C018  B9 00 01       MORE     LDA   STWORK,Y      ; load each ASCII byte of string
C01B  F0 06                   BEQ   OUT           ; if zero byte, we're finished
C01D  [redacted] D2 FF        JSR   CHROUT        ; otherwise, print it
C020  C8                      INY                 ; next byte
C021  D0 F5                   BNE   MORE          ; branch always
C023  60             OUT      RTS                 ; return to MAIN
                                                  ;
C024  89 B6 A0       FPNUM    .BYTE 137,182,160,0,0,0
                                                  ; the value for 365.25 in FP accumulator 1
```

See also BYTASC, CNUMOT, FACPRD, NUMOUT.

Name

Retrieve from expansion RAM memory

Description

FETCH is just the opposite of **STASH**; it tranfers bytes from expansion RAM in the model 1700 and 1750 RAM Expansion Modules into system memory.

Prototype

1. Enter this routine with the REC registers set with the appropriate system memory base address, expansion RAM base address, and number of bytes to transfer. The .X register should contain the system bank number.
2. Load .Y with the value required in the command register (location 57089) to perform a fetch operation.
3. JMP to the Kernal routine DMACALL.

Explanation

Memory locations 57088–57098, on the 128, are used to address the REC (RAM Expansion Controller chip) registers in the model 1700 or 1750 RAM Expansion Modules. The REC chip performs four different memory-management operations: stashing, fetching, swapping, and verifying.

The program below is designed to be used with the program provided with **STASH**. That particular program stores BASIC programs into one of four 32K memory partitions in the RAM expansion unit. This program, on the other hand, retrieves BASIC programs which have been stored to the expansion module.

So, after you've run the program associated with **STASH** and saved a few BASIC programs to expansion RAM, run this one. Notice that since it's assembled at a different location than its companion program, both can reside in memory simultaneously.

Next, SYS to the starting location (4864) of the program, following the SYS address with the number of a partition that contains a previously stored BASIC program. For example, suppose you wanted to fetch a previously saved BASIC program from partition 2, you'd enter **SYS4864,2**. The BASIC program in partition 2 would then be restored to the BASIC text area.

The program associated with **STASH**, when called, saves the BASIC pointers—the start- and end-of-BASIC addresses—followed by the BASIC program itself. Two separate transfer operations are required to restore it. The BASIC pointers are

the first thing brought back from the designated partition. Once they're installed, the BASIC program which follows is retrieved. As with the companion program, the expansion-RAM base address updates automatically with each byte transferred (bits 6 and 7 in 57098 are 00 by default).

Routine

```
1300                CHROUT   =    65490
1300                DMACALL  =    65360         ; Kernal routine which passes command in .X
                                                ; to DMA controller
1300                DMASYA   =    57090         ; DMA system memory base address register
1300                DMAEXA   =    57092         ; DMA expansion memory base address
                                                ; register
1300                DMABNK   =    57094         ; DMA expansion memory bank register
1300                DMADAT   =    57095         ; DMA number of bytes to transfer
1300                TXTTAB   =    45            ; start-of-BASIC pointer
1300                TEXTTP   =    4624          ; end-of-BASIC program pointer
1300                ZP       =    251
                                                ;
                                                ; Get BASIC program from RAM expansion
                                                ; bank 0 or 1 on 32K boundaries.
                                                ; Use this program in tandem with the
                                                ; program under STASH.
1300  C9 01                  CMP  #1            ; make sure .A is in range 1-4
1302  90 5D                  BCC  PRTMSG        ; .A is less than 1, so print an error message
                                                ; and leave
1304  C9 05                  CMP  #5
1306  B0 59                  BCS  PRTMSG        ; .A is 5 or greater, so print error message
                                                ; and leave
1308  38                     SEC                ; now subtract 1 to put it in range 0-3
1309  E9 01                  SBC  #1
130B  4A                     LSR                ; determine RAM expansion bank
130C  8D 06 DF               STA  DMABNK        ; store it into register
130F  A9 00                  LDA  #0            ; determine 32K offset in each bank (high
                                                ; byte)
1311  8D 04 DF               STA  DMAEXA        ; also store zero into base address for
                                                ; expansion memory (low byte)
1314  90 02                  BCC  EXPOFF        ; if partition number is 1 or 3, carry is clear,
                                                ; so 0K offset
1316  A9 20                  LDA  #32           ; offset by 32K if partition number is 2 or 4
1318  8D 05 DF  EXPOFF       STA  DMAEXA+1      ; store in base address for expansion memory
                                                ; (high byte)
131B  A9 FB                  LDA  #ZP           ; store starting address of two pointers in
                                                ; system-memory address register
131D  8D 02 DF               STA  DMASYA        ; low byte
1320  A9 04                  LDA  #4            ; store number of bytes to transfer in DMA
                                                ; register (low byte)
1322  8D 07 DF               STA  DMADAT
1325  A9 00                  LDA  #0            ; store zero to high byte
1327  8D 08 DF               STA  DMADAT+1
132A  8D 03 DF               STA  DMASYA+1      ; also store zero to high byte of system-
                                                ; memory address
132D  AA                     TAX                ; put system-memory bank number in .X
132E  20 6F 13               JSR  FETCH         ; retrieve BASIC pointers
1331  A5 FB                  LDA  ZP            ; install start-of BASIC pointer
1333  85 2D                  STA  TXTTAB
1335  A5 FC                  LDA  ZP+1
1337  85 2E                  STA  TXTTAB+1
1339  A5 FD                  LDA  ZP+2          ; install end-of-BASIC pointer
133B  8D 10 12               STA  TEXTTP
133E  A5 FE                  LDA  ZP+3
```

```
1340  8D 11 12            STA   TEXTTP+1
                                              ; Now retrieve BASIC program which was
                                              ; saved after the pointers
1343  38                  SEC                 ; determine number of bytes in BASIC
                                              ; program
1344  AD 10 12            LDA   TEXTTP        ; get end-of-BASIC low byte
1347  E5 2D               SBC   TXTTAB        ; subtract start-of BASIC low byte
1349  8D 07 DF            STA   DMADAT        ; store result into DMA register for number of
                                              ; bytes to transfer
134C  AD 11 12            LDA   TEXTTP+1      ; get end-of-BASIC high byte
134F  E5 2E               SBC   TXTTAB+1      ; subtract start-of-BASIC high byte
1351  8D 08 DF            STA   DMADAT+1      ; store to high byte of register
1354  A5 2D               LDA   TXTTAB        ; store starting address of BASIC as system
                                              ; base address
1356  8D 02 DF            STA   DMASYA
1359  A5 2E               LDA   TXTTAB+1
135B  8D 03 DF            STA   DMASYA+1
                                              ; System bank number is in .X, and DMAEXA
                                              ; updates automatically (see 57098).
135E  4C 6F 13            JMP   FETCH         ; retrieve BASIC program and RTS
                                              ;
1361  A0 00      PRTMSG   LDY   #0            ; index for PRTLOP
1363  B9 74 13   PRTLOP   LDA   ERRMSG,Y      ; get a character for the error message
1366  F0 06               BEQ   PRTEND        ; end on a zero byte
1368  20 D2 FF            JSR   CHROUT        ; print the character if not zero
136B  C8                  INY                 ; next character
136C  D0 F5               BNE   PRTLOP        ; branch always
136E  60         PRTEND   RTS                 ; leave the program
                                              ;
                                              ; Enter this routine with DMA registers set
                                              ; up, and system bank number in .X
136F  A0 81      FETCH    LDY   #%10000001    ; command register (57089) value for fetch
1371  4C 50 FF            JMP   DMACALL       ; call DMA Kernal routine and RTS
                                              ;
1374  4E 4F 54   ERRMSG   .ASC  "NOT A VALID PARTITION NUMBER"
                                              ; error message
1390  00                  .BYTE 0             ; terminator byte
```

See also STASH.

Name

General memory fill

Description

This routine fills a portion of memory with a particular byte. Just specify in the equates the starting address for the portion of memory you want to fill (BLOCK), the number of bytes you want to fill (NUMBER), and the particular byte you want to store (FILBYT).

Prototype

1. Store the number of bytes to be filled into the variable COUNTR at the end of the program.
2. Store the accumulator containing the fill byte into a temporary storage location (TEMPA).
3. In FILLOP, store the contents of TEMPA in BLOCK, using zero-page addressing until COUNTR decrements to zero. Then return to the calling program.

Explanation

To demonstrate **FILMEM**, the example program stores a 90 (screen code for Z) into 400 bytes of screen memory.

Within the routine itself, a two-byte counter (COUNTR) decrements each time a byte is copied. When this counter reaches 0 (the high byte must decrement to 255 on the last pass), the routine is complete.

On the 128, to fill memory in another bank, use the Kernal routine INDSTA at 65399. Define the target bank number at the end of the program as BNKFIL. Then substitute the four commented instruction lines in the middle of **FILMEM** for the STA (ZP),Y at $C024.

Routine

```
C000                  ZP      =     251
C000                  CHROUT  =     65490
C000                  BLOCK   =     1384       ; memory block to fill
C000                  FILBYT  =     90         ; byte to fill with
C000                  NUMBER  =     400        ; number of bytes to fill
C000                  INDSTA  =     65399      ; Kernal routine to store indirectly to any
                                               ; bank (128 only)
                                               ;
                                               ; Fill NUMBER of bytes of memory with the
                                               ; value in .A.
C000  A9 93           CLRCHR  LDA   #147       ; clear the screen
C002  20 D2 FF                JSR   CHROUT
C005  A9 68                   LDA   #<BLOCK    ; store memory block to fill in zero page, low
                                               ; byte first
C007  85 FB                   STA   ZP
C009  A2 05                   LDX   #>BLOCK    ; then high byte
C00B  86 FC                   STX   ZP+1
```

```
C00D  A2 90              LDX  #<NUMBER    ; then put low byte of number of bytes to fill
                                          ; in .X
C00F  A0 01              LDY  #>NUMBER    ; and high byte in .Y
C011  A9 5A              LDA  #FILBYT     ; byte to fill with in .A (screen code for a
                                          ; diamond)
C013  4C 16 C0           JMP  FILMEM      ; fill memory and RTS
                                          ;
                                          ; Fill memory. Enter with the number of
                                          ; bytes to move in .X (low)
                                          ; and .Y (high). Memory block is in two bytes
                                          ; at ZP
C016  8E 3C C0  FILMEM   STX  COUNTR      ; store number to COUNTR, low byte first
C019  8C 3D C0           STY  COUNTR+1    ; then high byte
C01C  8D 3E C0           STA  TEMPA       ; store FILBYT temporarily
C01F  A0 00              LDY  #0          ; index for FILLOP
C021  AD 3E C0  FILLOP   LDA  TEMPA       ; restore FILBYT in .A
C024  91 FB              STA  (ZP),Y      ; store a byte into memory block
                                          ; For the 128, substitute the next four lines
                                          ; for the previous line
                                          ; to fill memory in another bank.
                                          ; LDX #ZP; put zero-page pointer to
                                          ; memory block in location 697
                                          ; STX 697
                                          ; LDX BNKFIL; bank number for memory
                                          ; fill
                                          ; JSR INDSTA; store into bank .X
                                          ; beginning at block
                                          ;
C026  E6 FB              INC  ZP          ; increase ZP pointer by one, low byte first
C028  D0 02              BNE  DECCTR      ; if low byte hasn't turned over, decrement
                                          ; the counter
C02A  E6 FC              INC  ZP+1        ; increase ZP high byte
C02C  CE 3C C0  DECCTR   DEC  COUNTR      ; decrement counter low byte
C02F  D0 F0              BNE  FILLOP      ; if low byte hasn't turned over, continue
                                          ; filling
C031  CE 3D C0           DEC  COUNTR+1    ; otherwise, decrement the high byte
C034  AD 3D C0           LDA  COUNTR+1    ; determine whether we've filled the last
                                          ; page
C037  C9 FF              CMP  #255        ; on the last page, high byte of counter goes
                                          ; from 0 through 255
C039  D0 E6              BNE  FILLOP      ; if not on the last page, continue
C03B  60                 RTS
                                          ;
C03C  00 00     COUNTR   .WORD 0          ; two-byte counter for remaining number of
                                          ; bytes to fill
C03E  00        TEMPA    .BYTE 0          ; temporary .A storage
                                          ; BNKFIL .byte 15, the bank number for
                                          ; memory fill (128 only)
```

Name

Find the cursor location

Description

FINDCR uses the Kernal routine PLOT to return the current cursor position by row (in .X) and column (in .Y). This routine is handy in game writing, especially when you're tracking a player's screen position.

Prototype

1. Set the carry flag—required by PLOT.
2. JSR to the Kernal routine PLOT and return (or simply JMP to PLOT).

Explanation

The example routine allows you to move about the screen by using the cursor keys or simply by typing in characters. Whenever you press X, its position is returned to the main program by **FINDCR**. The row and column number, separated by a space, are then printed with **NUMOUT**.

Note: Setting the carry flag and calling PLOT causes the cursor position to be placed in .X and .Y. Upon returning from PLOT, .X contains one fewer than the actual row number, while .Y contains one fewer than the column number—that is, if you're used to numbering the columns 1–40 and the rows 1–25. Programmers who start counting at zero will find the columns and rows to be just right (0–39 and 0–24, or 0–79 and 0–24 on the 80-column screen of the 128). If you're working within a window on the 128, the values returned in .X and .Y are relative to the top of the window rather than to the top of the screen.

Warning: If you use this routine within a loop indexed by .Y or .X, be sure to save the current index value to a safe location before calling it since PLOT affects both the .X and .Y registers.

Routine

```
C000                PLOT    =    65520     ; Kernal cursor-position routine
C000                GETIN   =    65508
C000                CHROUT  =    65490
C000                LINPRT  =    48589     ; LINPRT = 36402 on the 128
                                           ;
                                           ; Print the current cursor row (0-24) and
                                           ; column (0-39) when X key is pressed.
C000  A9 93         CLRCHR  LDA  #147      ; clear the screen
C002  20 D2 FF              JSR  CHROUT
C005  20 E4 FF      LOOP    JSR  GETIN     ; get a character
C008  C9 58                 CMP  #88       ; is it X?
```

```
C00A  F0  06             BEQ   LOCATE    ; it's X, so determine position
C00C  20  D2  FF         JSR   CHROUT    ; otherwise, print character
C00F  4C  05  C0         JMP   LOOP      ; and continue
C012  20  35  C0  LOCATE JSR   FINDCR    ; determine the cursor position
C015  8C  3A  C0         STY   TEMPY     ; save .Y
C018  A9  58             LDA   #"X       ; print X
C01A  20  D2  FF         JSR   CHROUT
C01D  A9  0D             LDA   #13       ; print RETURN
C01F  20  D2  FF         JSR   CHROUT
C022  20  30  C0         JSR   NUMOUT    ; print the row
C025  A9  20             LDA   #32       ; print space
C027  20  D2  FF         JSR   CHROUT
C02A  AE  3A  C0         LDX   TEMPY     ; get the column
C02D  4C  30  C0         JMP   NUMOUT    ; print the column and RTS
                                         ;
                                         ; Print the two-byte integer in .X (low byte)
                                         ; and .A (high byte).
C030  A9  00      NUMOUT LDA   #0        ; high byte of row or column is always zero
                                         ; here
C032  4C  CD  BD         JMP   LINPRT    ; print number and RTS
                                         ;
                                         ; Locate the cursor. Return position in .X
                                         ; (row) and .Y (column).
C035  38          FINDCR SEC             ; set carry to locate cursor
C036  20  F0  FF         JSR   PLOT      ; locate the cursor
C039  60                 RTS
                                         ;
C03A  00          TEMPY  .BYTE 0         ; temporary .Y storage
```

See also PLOTCR.

Name

Find the program counter address (from a subroutine)

Description

The program counter (PC) is an internal register in the 6510 and 8502 microprocessors that keeps track of which ML instruction is currently being executed. There are times when it's necessary to find out where in memory a program is located. And, on occasion, a subroutine may need to figure out which part of the program did the original JSR. This subroutine figures out the program counter address and stores it in memory.

Prototype

1. JSR to **FINDME** (the subroutine that finds the PC).
2. Within the subroutine, use PLA twice to pull the two-byte address off the stack.
3. After storing the address somewhere, push the address back.
4. RTS.

Explanation

When you JSR (Jump to a SubRoutine), the computer has to be able to figure out the return address when an RTS (ReTurn from Subroutine) instruction ends the subroutine. So, just before jumping to the subroutine, the computer puts the return address on the stack, high byte first, followed by the low byte.

Knowing this makes it a simple matter to pull the address from the stack and store it in memory (location 829 was chosen for storage, for no particular reason except that it's available on the 64). Before the subroutine executes the RTS to get back, you must put the return address back on the stack so that the RTS will work properly.

Note: The main program that calls **FINDME** does the JSR at location $C000. The return address should bring you back to $C003, the next instruction after JSR **FINDME**. Actually, the address that's pushed onto the stack is $C002 (the return address minus one). What happens during an RTS is that the address is taken from the stack and then the PC is incremented. After each instruction, the program counter counts forward, and RTS is no exception. Thus, when the address is printed, you'll see a 49154 (decimal) instead of a 49155.

Warning: This might seem to be a convenient way to figure out the program counter value, in case you want to relocate the routine to another place in memory. The problem is that JSR uses an absolute address, so the **FINDME** subroutine must be at a known location. If you relocate the object code to $8000, for example, the first three bytes of the program (20 0D C0) will still JSR to $C00D. You should either load the **FINDME** routine as a separate program or limit its use to finding the address of the calling routine. Another routine (**FINDPC**) may be preferable if you're moving ML routines around and don't know where they'll be placed.

Location 829 is not available on the 128. Programmers should substitute two other consecutive free memory locations on the 128.

Routine

```
C000                IMHERE   =     829          ; choose a different address for the 128
C000                LINPRT   =     $BDCD        ; general routine to print a two-byte unsigned
                                                ; integer
                                                ; LINPRT = $8E32 ; (substitute this for
                                                ; the 128)
                                                ;
C000  20  0D  C0             JSR   FINDME
                                                ; Now print address value.
C003  AE  3D  03             LDX   IMHERE       ; low byte
C006  AD  3E  03             LDA   IMHERE+1     ; high byte
C009  20  CD  BD             JSR   LINPRT       ; print as a decimal number
C00C  60                     RTS                ; all done
                                                ;
                                                ; subroutine to find address (minus 1) of
                                                ; calling routine
C00D  68            FINDME   PLA                ; pull low byte from stack
C00E  8D  3D  03             STA   IMHERE       ; store in IMHERE
C011  68                     PLA                ; now get the high byte
C012  8D  3E  03             STA   IMHERE+1     ; and store in IMHERE+1
C015  48                     PHA                ; put it back
C016  AD  3D  03             LDA   IMHERE       ; low byte
C019  48                     PHA                ; goes back also
C01A  60                     RTS                ; otherwise, this RTS won't work
```

See also FINDPC.

Name

Find the program counter address (in-line code)

Description

Most ML instructions are location-independent. LDA #$08 loads an 8 into the accumulator regardless of where the instruction happens to reside in memory. But JMPs and JSRs are absolute instructions. If a program is relocated to a new section of memory, the internal JMPs and JSRs should be modified. This routine lets you find out where you are in memory, so you may make the necessary modifications.

Prototype

1. Put an RTS instruction somewhere safe in memory.
2. JSR to it, which means coming back immediately.
3. Transfer the stack pointer to .X.
4. Decrement .X twice and put the result back in the stack pointer with TXS.
5. Pull the address from the stack.

Explanation

The JSR instruction pushes the return address (minus 1) onto the stack, which for the 6510 and 8502 microprocessors is always located in page 1. The stack builds down in memory from $1FF.

The opcode for the RTS instruction is 96 (decimal), so if a 96 is stored in memory and you JSR there, the program bounces right back to where it started. But in the meantime, the stack has very briefly held the return address from the JSR. All you have to do to reset the stack pointer to the address is transfer the stack pointer to .X (TSX), decrement .X twice, and transfer .X back to the stack pointer (TXS). PLA pulls the low byte off the stack and another PLA pulls the high byte.

Note: The resulting address is stored in the IMHERE location (location 829 is used in the example routine, but it's available only on the 64). The JSR at $C005 originally put the address on the stack. The return address is $C008, but the value on the stack is actually one less than that. When the transfers, decrements, and pulls are finished, the result will be a $07 in IMHERE and a $C0 in IMHERE+1.

Warning: Do *not* use this as a subroutine. If you do, you'll find the address of the subroutine instead of the routine that called it.

Locations 828-830 are not free on the 128. Substitute three other free locations for the labels FREE and IMHERE on the 128.

Routine

```
C000                FREE      =    828        ; could be any free location
C000                IMHERE    =    829        ; two bytes to store eventual program counter
                                              ; (choose other addresses for the 128)
C000  A9 60         FINDPC    LDA  #96        ; the object code for RTS
C002  8D 3C 03                STA  FREE       ; set up the shortest subroutine, there and
                                              ; back
C005  20 3C 03                JSR  FREE       ; bouncing back (note the address)
                                              ;
C008  BA            MINUS     TSX             ; stack pointer in X
C009  CA                      DEX             ; decrement once
C00A  CA                      DEX             ; and twice
C00B  9A                      TXS             ; put it back in the stack pointer
C00C  68                      PLA             ; pull one byte
C00D  8D 3D 03                STA  IMHERE     ; low byte of PC into IMHERE
C010  68                      PLA             ; pull the next byte
C011  8D 3E 03                STA  IMHERE+1   ; high byte of PC
C014  60                      RTS             ; end of this routine, normally we'd process
                                              ; address value
                                              ; The value in 829 will point to
                                              ; one byte before the label MINUS above.
```

See also FINDME.

Name

Read a joystick fire button

Description

This simple routine checks the fire button of the specified joystick.

Prototype

1. Enter this routine with the accumulator containing the number of the joystick whose fire button you wish to check.
2. Load the contents of the appropriate joystick register into the accumulator.
3. Test bit 4 of the accumulator by ANDing with %00010000 and RTSing to the main program. (If the zero flag is set as a result of the AND, the fire button is pressed.)

Explanation

Pressing the fire button on either joystick clears bit 4 of the corresponding joystick register. Joystick port 1 is wired to the register at 56321 (CIAPRB), while port 2 is connected to 56320 (CIAPRA). You might expect the sequence of the registers (56320–56321) to be the same as the sequence of the joystick labels (1–2), but for some reason they're switched.

Before you call **FIREBT**, provide the joystick number in the accumulator. The routine then reads the appropriate register and returns with the zero flag set if the fire button for that joystick is being pressed.

In the example program, pressing the fire button on joystick 1 causes the border color of the screen to increment.

Routine

```
C000                CIAPRA    =     56320         ; data port register A
C000                BGCOL0    =     53281         ; screen background color register
                                                  ;
                                                  ; Read joystick 1 fire button. Change screen
                                                  ; color when pressed.
C000  A9  01        JOYLOP    LDA   #1            ; put joystick number in .A
C002  20  0B  C0              JSR   FIREBT        ; read fire button
C005  D0  F9                  BNE   JOYLOP        ; if fire button not pressed, check it again
C007  EE  21  D0              INC   BGCOL0        ; increment screen color
C00A  60                      RTS                 ; and you're done
                                                  ;
                                                  ; Enter the routine with joystick number
                                                  ; in .A.
C00B  29  01        FIREBT    AND   #1            ; determine joystick offset
C00D  AA                      TAX                 ; put offset in .X
C00E  BD  00  DC              LDA   CIAPRA,X      ; read joystick 1 (.X = 1) or 2 (.X = 0)
C011  29  10                  AND   #%00010000    ; test fire button bit—result is zero if fired
C013  60                      RTS                 ; zero flag set if fired
```

See also JOY2TO, JOY2SE, JOYSTK.

Name

Format a disk

Description

A disk must be formatted before it can be used. This process lays down the tracks and sectors that will later hold the programs and files you save to the disk. This routine formats a disk, preparing it for reading and writing.

Prototype

1. Open the command channel (with the Kernal SETLFS, SETNAM, and OPEN routines).
2. Send the command "N0:*diskname, ID*".
3. Close the file.

Explanation

This routine is the equivalent of the BASIC command OPEN 1, 8, 15, "N0:*diskname,ID*":CLOSE 1. The first number (the logical file number) is unimportant. The second is the disk drive number, which is almost always 8 unless you own more than one drive, in which case the device number may be 8–11. The final number is the secondary address. When you open a disk file, the secondary address is the channel number, and channel 15 is reserved for direct commands to the drive. The *N0:* command is short for NEW the disk. It's followed by your choice of disk name, plus the ID.

Note: If the disk has previously been formatted, you can omit the ID number. The disk will not be reformatted. Instead, the directory will be cleared and the disk will be renamed with the new disk name. As far as the disk drive is concerned, this is equivalent to reformatting the disk. Leaving off the ID speeds up the formatting process.

Warning: This program will erase everything on your disk. Experiment with it at your own risk.

Routine

```
C000                 SETLFS   =    $FFBA
C000                 SETNAM   =    $FFBD
C000                 OPEN     =    $FFC0
C000                 CLOSE    =    $FFC3
C000                 CLRCHN   =    $FFCC

C000  A9 01          FORMAT   LDA  #1          ; logical file (1)
C002  A2 08                   LDX  #8          ; disk drive is device 8
C004  A0 0F                   LDY  #15         ; command channel 15
C006  20 BA FF                JSR  SETLFS      ; prepare to open it
C009  A9 0D                   LDA  #BUFLEN     ; length of buffer
C00B  A2 1E                   LDX  #<BUFFER    ; X and Y hold the
```

```
C00D  A0  C0              LDY   #>BUFFER     ; address of the buffer
C00F  20  BD  FF          JSR   SETNAM       ; set name
C012  20  C0  FF          JSR   OPEN         ; open it
C015  A9  01              LDA   #1           ; and immediately
C017  20  C3  FF          JSR   CLOSE        ; close the command channel
C01A  20  CC  FF          JSR   CLRCHN       ; clear the channels
C01D  60                  RTS                ; all done
                                             ;
                                             ; Data area
C01E  4E  30  3A  BUFFER  .ASC  "N0:MYDISK,MD"
                                             ; Substitute your own name for MYDISK, and
                                             ; your own ID for MD
C02A  0D                  .BYTE 13           ; RETURN character
C02B              BUFLEN  =     * - BUFFER
```

See also CONCAT, COPYFL, INITLZ, RENAME, SCRTCH, VALIDT.

Name

Print the number of free sectors remaining on the disk

Description

FRESEC prints the number of free sectors remaining on the disk without printing the entire directory. Such a routine is useful in reporting to the user the amount of space remaining on the disk before a save is attempted.

Prototype

1. On the 128, set the bank to 15.
2. OPEN 1,8,0 with a directory specifier, *$0:*, and a nonexistent filename (SETLFS, SETNAM, and OPEN) for reading.
3. On the 128, prior to SETNAM, load .X with the bank containing the directory filename. Then JSR to SETBNK.
4. Read in and discard the first six bytes (two track and sector bytes, two link bytes, and two for the number of blocks occupied).
5. Read bytes from the disk header until a zero byte occurs.
6. Discard the two link bytes from the BLOCKS FREE entry.
7. Print the two-byte number representing the blocks free with **NUMOUT**.
8. Print the BLOCKS FREE message and close the file.

Explanation

In **FRESEC**, we use the directory name *$0:Z-£=U*. This tells the computer to search the directory for any USR programs that begin with the characters Z-£. Of course, it's very unlikely that such a file exists. Not finding this filename, the computer loads the directory header and reports the number of free blocks on the disk.

To see what we mean, try this from BASIC: Just LOAD "$0:Z-£=U",8 and list what loads.

The directory file is structured much like a BASIC program file. Within the directory, each entry (including the disk header and the BLOCKS FREE message) is comparable to a program line.

At the beginning of the directory are two bytes that act as a load address for a program. (If you LOAD "$",8,1, the directory finds its way into 1024, which is where screen memory is located.) We have no use for these bytes, and they are discarded. The next two bytes are link bytes that point to the address of the first entry in the directory. These are equivalent to

the link bytes in a BASIC program file that point to the next program line. Again, these bytes, here associated with the disk name, are discarded.

The next two bytes represent the number of blocks occupied by that particular program entry (or filename). If the entry is the disk header, these two bytes are always zero, and we discard them.

After this, we move to the end of the disk header description by finding the next zero byte. Just as with a BASIC program, this zero byte marks the end of each line (or entry). So now, we're positioned at the beginning of the BLOCKS FREE entry. Again, the first two bytes in the entry are link bytes, and we ignore them.

Finally, we've reached our destination within the directory. The next two bytes represent the number of free sectors remaining on the disk in low-byte/high-byte form. This two-byte integer is printed out with **NUMOUT**, a space is inserted, and BLOCKS FREE is printed.

Our purpose accomplished, file 1 is closed and default devices are restored with CLRCHN.

Note: **FRESEC** currently lacks disk error checking. You can easily add this feature, if you like, by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM, as noted in the source listing. Jump to **DERRCK** immediately after you have opened file 1 to the disk. Also, be sure to open the error channel (15) at the beginning of the program. (Again, this is noted in the source listing.)

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000          SETLFS   =   65466      ; (128 only)
C000          SETNAM   =   65469
C000          OPEN     =   65472
C000          CHKIN    =   65478
C000          CHRIN    =   65487
C000          CHROUT   =   65490
C000          CLOSE    =   65475
C000          CLRCHN   =   65484
C000          LINPRT   =   48589      ; LINPRT = 36402 on the 128
C000          SETBNK   =   65384      ; Kernal bank number for OPEN and
                                      ; filename (128 only)
C000          MMUREG   =   65280      ; MMU configuration register (128
                                      ;
                                      ; Read and print the number of free sectors
                                      ; remaining on the disk
                                      ;
                                      ; Open channel 15 here if you include disk
```

```
                                              ; error checking (DERRCK).
                                              ;
C000                FRESEC  =     *
                                              ; LDA #0; set the 128 to bank 15 (128 only)
                                              ; STA MMUREG; (128 only)
C000  A9 01                 LDA   #1          ; logical file 1
C002  A2 08                 LDX   #8          ; device number for disk drive
C004  A0 00                 LDY   #0          ; secondary address to read
C006  20 BA FF              JSR   SETLFS      ; set file parameters
                                              ; Include the following three instructions
                                              ; for the 128 only.
                                              ; LDA BNKNUM; bank number for data
                                              ; LDX BNKFNM; bank containing the
                                              ; ASCII filename
                                              ; JSR SETBNK
C009  A9 08                 LDA   #FNLENG     ; length of filename
C00B  A2 51                 LDX   #<FILENM    ; address of filename
C00D  A0 C0                 LDY   #>FILENM
C00F  20 BD FF              JSR   SETNAM      ; set up filename
C012  20 C0 FF              JSR   OPEN        ; open the directory file for reading
                                              ;
                                              ; JSR DERRCK; Insert here for disk error
                                              ; checking
                                              ;
C015  A2 01                 LDX   #1
C017  20 C6 FF              JSR   CHKIN       ; take input from file 1
C01A  A2 05                 LDX   #5          ; discard six bytes (track and sector, link,
                                              ; and blocks occupied—two each)
C01C  20 CF FF     TOSSIT   JSR   CHRIN
C01F  CA                    DEX
C020  10 FA                 BPL   TOSSIT
                                              ;
                                              ; Read information on disk header until
                                              ; zero byte is reached. What follows
                                              ; is the number of blocks occupied (two
                                              ; bytes) and BLOCKS FREE message.
C022  20 CF FF     INLOOP   JSR   CHRIN       ; get a byte from open file
C025  D0 FB                 BNE   INLOOP      ; is it a zero byte yet?
                                              ; We've reached the end of the header. The
                                              ; next two bytes are link bytes,
C027  20 CF FF              JSR   CHRIN       ; discard them
C02A  20 CF FF              JSR   CHRIN
                                              ;
                                              ; Print the two-byte number representing
                                              ; number of blocks remaining with
                                              ; NUMOUT.
C02D  20 CF FF              JSR   CHRIN
C030  AA                    TAX               ; low byte of number
C031  20 CF FF              JSR   CHRIN       ; high byte of number
C034  20 CD BD     NUMOUT   JSR   LINPRT      ; print the number
                                              ;
                                              ; Print BLOCKS FREE message.
C037  A9 20                 LDA   #32         ; print a SPACE
C039  20 D2 FF              JSR   CHROUT
C03C  20 CF FF     PRTLOP   JSR   CHRIN       ; get a character
C03F  20 D2 FF              JSR   CHROUT      ; and print it
C042  D0 F8                 BNE   PRTLOP      ; if not zero byte, get another character
C044  A9 0D                 LDA   #13         ; last character, so print a RETURN
C046  20 D2 FF              JSR   CHROUT
C049  A9 01                 LDA   #1
C04B  20 C3 FF              JSR   CLOSE       ; close file 1
```

```
C04E  4C  CC  FF            JMP   CLRCHN       ; clear all channels, restore default devices,
                                                ; and return
                                                ;
                                                ; insert DERRCK routine here if you're
                                                ; including error checking.
                                                ;
C051  24  30  3A  FILENM    ASC   "$0:Z-£=U"   ; filename for USR program Z-£ in the
                                                ; directory
C059              FNLENG    =     * - FILENM   ; length of filename
                                                ; Include the next two variables on the 128.
                                                ; BNKNUM .BYTE 0; bank number for data
                                                ; BNKFNM .BYTE 0; bank number where
                                                ; ASCII filename is located
```

See also DIRBYT, DIRPRG.

Name

Exit machine language and GOTO a BASIC line number

Description

There are several ways to combine machine language routines with BASIC. The most common way to call an ML program is with the SYS statement. When you're finished, RTS returns control to the BASIC program.

With the following **GOTOBL** routine, a machine language program can return to any given line number within a BASIC program. This means that if you SYS to an ML routine, you can return to the BASIC program at some point other than where you SYSed from. If you want, you can even have a series of conditional GOTOs to different BASIC line numbers within your ML program. Or you can pass a variable value to the ML routine using **PASFMV**, convert it to an integer, and GOTO the chosen line number.

Prototype

1. Store the BASIC line number you intend to go to at the end of the routine (BSLINE).
2. Within the routine itself, store the low and high bytes of the target BASIC line number in .A and .Y and store them in LINNUM.
3. Then jump into BASIC's GOTO routine.

Explanation

In the example program, **GOTOBS** performs a GOTO to line 2000 within the BASIC program. When you try the program, be sure that you have a line 2000 in memory; otherwise, you'll get an undefined line error.

GOTOBL itself is very straightforward. Within it, the target line number (BSLINE) is placed in the two-byte LINNUM (location 20 on the 64 and 22 on the 128). After this, the program jumps directly into the middle of BASIC's GOTO routine (43196 on the 64 and 23035 on the 128). We skip the part of the GOTO routine that gets the target line number since it's already provided.

Routine

```
C000                LINNUM   =     20           ; LINNUM = 22 on the 128—integer line
                                                ; number
C000                GOTOBS   =     43196        ; GOTOBS = 23035 on the 128—GOTO the
                                                ; line number in LINNUM
                                                ;
                                                ; Exit ML and GOTO a BASIC line
C000  AD 0D  C0  GOTOBL   LDA   [redacted]      ; store low byte of line number to return to
C003  85  14              STA   LINNUM
C005  AC  0E  C0          LDY   BSLINE+1        ; now, store high byte
C008  84  15              STY   LINNUM+1
C00A  4C  BC  A8          JMP   GOTOBS          ; exit ML, GOTO BASIC line
                                                ;
C00D  D0  07     BSLINE   .WORD 2000            ; BASIC line to GOTO
```

See also PASFMV, PASMEM, PASREG, PASUSR.

Name

GOTO from a character input using sequential compares and branches

Description

This is probably the fastest way to execute a routine based on a limited number of keyboard responses. Here, you simply get a character from the keyboard and check the response sequentially against a series of allowed ASCII responses. If a suitable response is found, branch to the appropriate routine.

Prototype

1. Get a keypress.
2. Compare its ASCII value with each acceptable response and branch to the appropriate routine.
3. If the response is not among those compared with, branch to step 1.

Explanation

The example program illustrates a common programming situation—checking for a Y (yes) or N (no) response. If you press Y, the screen border color changes to white. An N changes it to black.

As it's currently written, the routine checks for two characters. But additional CMP #*ASCII value*:BEQ *routine address* instructions can be added if you need to check for more keys.

If many characters are checked for, place the CMP and BEQ steps for the most commonly pressed keys early in the code. This will speed execution of the routine slightly.

If the routines you wish to execute lie outside the range of the branch instruction (128 bytes backward or 127 bytes forward), you can use **GOTOST**. Or, you can use a CMP #*ASCII value*: BNE *next compare*: JMP *routine address* arrangement instead.

Routine

```
C000                GETIN    =    65508
C000                EXTCOL   =    53280      ; border color register
                                             ;
                                             ; Limit input to Y or N. Then, go to
                                             ; appropriate routine.
C000  20  E4  FF    GOTOCP   JSR  GETIN      ; get a character from keyboard
C003  C9  4E                 CMP  #78        ; is it N?
C005  F0  09                 BEQ  ROUTEN     ; N was pressed, so go to NO routine
C007  C9  59                 CMP  #89        ; is it Y?
```

```
C009  D0 F5             BNE  GOTOCP   ; neither N nor Y, so get another key
                                      ; If Y, fall through to ROUTEY
C00B  A9 01     ROUTEY  LDA  #1       ; Y routine
C00D  4C 15 C0          JMP  BORCOL   ; change border color to white
C010  A9 00     ROUTEN  LDA  #0       ; N routine
C012  4C 15 C0          JMP  BORCOL   ; change border color to black
                                      ;
                                      ; Set border color. Enter with color value
                                      ; in .A.
C015  8D 20 D0  BORCOL  STA  EXTCOL   ; set register
C018  60                RTS
```

See also GOTOST.

Name

GOTO from a character input and execute using the stack

Description

GOTOST, like **GOTOCP**, checks for limited keypresses, executing a certain routine based on the response. The approach taken here is preferred, however, when the number of keypresses and corresponding routines is lengthy.

As with **GOTOCP**, we begin by getting a character from the keyboard. At this point (in CHKLOP), we check the response against a number of suitable characters in a table (KEYS). If the incoming key is in the table, we go to the appropriate routine by placing its address, less one, on the stack and executing an RTS. The RTS causes the program to jump to the chosen routine.

The location of each acceptable routine is listed in a table of two-byte addresses (labeled ROUTES) at $C02C. These addresses are automatically calculated by the assembler.

Prototype

1. Get a keypress.
2. Check the key entered against a table of allowed character input.
3. If the input key is the same as a character in the table, use its relative position in the table to determine the address of the corresponding routine.
4. Push the high and low address bytes of the selected routine onto the stack.
5. Execute an RTS, thereby jumping to the chosen routine.

Explanation

The following program demonstrates this routine by checking for an A or a B keypress. If A is pressed, the background color of the screen is cycled through the available colors; if B is pressed, the border color rotates. If neither key is pressed, the program gets another keypress.

Note: The table of acceptable characters can contain the entire ASCII set (as many as 255 characters), if you like. To speed execution of the routine, place the characters representing the more likely responses at the beginning of the table.

Routine

```
C000          GETIN    =    65508
C000          BGCOL0   =    53281      ; text-screen background color register 0
C000          EXTCOL   =    53280      ; text-screen border color register
                                       ;
```

```
                                           ; Check for keys in table and execute
                                           ; appropriate routine using stack
                                           ; Change (A) background or (B) border color,
C000  4C 03 C0  LOOP     JMP   GOTOST      ; check for keys, and execute appropriate
                                           ; routine
                                           ;
C003  20 E4 FF  GOTOST   JSR   GETIN       ; get ASCII key value
C006  A2 00              LDX   #0
C008  DD 30 C0  CHKLOP   CMP   KEYS,X      ; check each character in table
C00B  F0 07              BEQ   FOUND       ; if found
C00D  E8                 INX
C00E  E0 02              CPX   #NUMKEY     ; check key number
C010  D0 F6              BNE   CHKLOP      ; if more in table, check next character
C012  F0 EF              BEQ   GOTOST      ; if no match, get another keypress
C014  8A        FOUND    TXA               ; character key has been pressed
C015  0A                 ASL               ; double its value since routines are at two-
                                           ; byte addresses
C016  AA                 TAX
C017  BD 2D C0           LDA   ROUTES+1,X  ; get high byte of routine address
C01A  48                 PHA               ; push it on stack
C01B  BD 2C C0           LDA   ROUTES,X    ; push low byte
C01E  48                 PHA
C01F  60                 RTS               ; RTS causes program to return to last
                                           ; address on stack plus one
                                           ;
                                           ; Routines for A and B follow.
C020  EE 21 D0  BCKCOL   INC   BGCOL0      ; cycle background color
C023  4C 00 C0           JMP   LOOP        ; and get another keypress
                                           ;
C026  EE 20 D0  BORCOL   INC   EXTCOL      ; cycle border color
C029  4C 00 C0           JMP   LOOP        ; and get another keypress
                                           ;
C02C  1F C0 25  ROUTES   .WORD BCKCOL  1,BORCOL-1
                                           ; two-byte addresses of each routine minus 1
C030  41 42     KEYS     .ASC  "AB"        ; list of acceptable keystrokes
C032            NUMKEY   =     *-KEYS      ; number of acceptable keys
```

See also GOTOCP.

Name

Hide a two-byte instruction with the BIT instruction

Description

The BIT instruction tests one value against another, but apart from setting a few status register flags, it changes the contents of neither the registers nor memory. Because it is almost a do-nothing command, BIT can be used to hide a two-byte instruction.

Prototype

1. Precede each instruction in a series with a .BYTE $2C.
2. Jump or branch into the list at various entry points.

Explanation

Suppose you saw the following fragment in a machine language routine. What would it do?

```
033C LDA #$41
033E BIT $42A9
0341 JSR $FFD2
```

If you enter it at $033C, the routine will put the ASCII value of *A* into the accumulator, perform a BIT, and then print the accumulator value. But what is the significance of the comparison with location $42A9? There is none. It doesn't matter what value is found at $42A9, and it doesn't matter that the N, Z, and V flags are affected by the BIT instruction.

Instead, the BIT instruction hides the two bytes $A9 and $42 (stored low byte first, of course). Those two bytes combine to form the instruction LDA #$42. So if you enter the routine just past the BIT instruction (at location $033F), the routine prints the letter *B*. As a single routine, it prints either an *A* or a *B*. There's no shorter way to write a two-in-one (or more) routine.

One valuable application for this little trick is in extending the range of branch instructions. A BEQ or BNE can branch forward 127 bytes or backward 128. But if you hide an additional BEQ or BNE inside a BIT, you can increase the range of a branch.

Routine

```
C000                 ZP       =    $FB
C000                 GETIN    =    $FFE4
C000                 CHROUT   =    $FFD2
                                                  ;
C000  20  E4  FF     ENTRY    JSR  GETIN          ; get a key
C003  F0  FB                  BEQ  ENTRY          ; go back if no key pressed
                                                  ;
```

```
C005  C9  31        HIDBIT  CMP   #49       ; the 1 key?
C007  F0  0D                BEQ   KEY1      ; branch ahead
C009  C9  32                CMP   [illegible]  ; is it a 2?
[illegible]  F0  0C         BEQ   KEY2      ; branch ahead
C00D  C9  33                CMP   #51       ; check for 3
C00F  F0  0B                BEQ   KEY3      ; yes, it is
C011  C9  34                CMP   #52       ; now a 4
C013  F0  0A                BEQ   KEY4      ; another branch
C015  2C                    .BYTE $2C       ; the BIT instruction
C016  A9  93        KEY1    LDA   #147      ; clear screen for 1
C018  2C                    .BYTE $2C
C019  A9  12        KEY2    LDA   #18       ; reverse on for 2
C01B  2C                    .BYTE $2C
C01C  E6  FB        KEY3    INC   ZP        ; another two-byte instruction for 3
C01E  2C                    .BYTE $2C
C01F  F0  06        KEY4    BEQ   QUIT      ; two bytes hiding another BEQ (always
                                            ; equal if we get here)
C021  20  D2  FF            JSR   CHROUT    ; print a key
C024  4C  00  C0            JMP   ENTRY     ; and jump back
C027  60            QUIT    RTS
```

Name

Fill high-resolution color memory

Description

In machine language, setting up a high-resolution graphics screen on the 64 or 128 is a multistep process. The 16K video bank where the screen is to be located is selected (**VIDBNK**), bitmap mode is enabled (**BITMAP**), and the newly created screen is cleared (**CLRHRS** or **CLRHRF**).

In addition, before you draw anything on this screen, the foreground color—the color of the individual pixels or dots on the screen—and the background color must be assigned. Just as **COLFIL** fills color memory for a text screen, **HRCOLF** fills the 1000-byte area of memory associated with the standard high-resolution screen (as opposed to a multicolor-mode screen).

Prototype

1. Enter this routine with the foreground color value in the accumulator and the background color value in .X.
2. Store the .X register contents into a temporary location.
3. Shift the low nybble of the accumulator into its high nybble.
4. OR in the temporary location so that the accumulator contains the foreground color in its high nybble and the background color in its low nybble.
5. Within a loop, store .A in all 1000 bytes representing high-resolution color memory and return to the main program.

Explanation

The example program sets up a high-resolution graphics screen (or bitmap) at the start of video bank 1—location 16384 to be exact. Color memory for this screen directly follows.

Placing the bitmap screen in a video bank other than bank 0 makes the code a little more involved, especially on the 128. Above 16383, memory in bank 15 on the 128 consists only of ROM, although POKEing values into locations 16384 or higher of bank 15 causes whatever is being stored to go into bank 0 RAM. And this, among other reasons, requires us to treat the 128 version differently, as you'll soon see. (For comparison purposes, you might look at the program under **CLRHRF**, which creates a high-resolution graphics screen at location 8192.)

Initially, on the 64, the contents of the VIC-II chip mem-

ory control register, or VMCSB at 53272, are saved to a temporary location. This register contains the offset address within the current video bank for the character set (in its low nybble) and the text screen (in its high nybble). By saving it out in this manner, we'll later be able to restore the text screen when we exit bitmap mode.

On the 128, you don't need to save VMCSB. The reason is that on this machine VMCSB takes its value during each IRQ interrupt from one of two shadow registers. In text mode, this register is VM1 at 2604, while in bitmap mode it's VM2 at 2605. Since we never alter VM1 in the program, we don't need to worry about storing it (or VMCSB).

Next, a value of %10000000 is stored into VMCSB (into VM2 on the 128, since this register gets copied to VMCSB when we enter bitmap mode). The high nybble of VMCSB (or VM2) still points to the offset address for the text screen, but in normal bitmap mode, the text screen is actually color memory for the graphics screen. So storing an $8 high nybble offsets color memory by 8K in the video bank we're about to choose (bank 1). This places color memory for the bitmap screen at 24576. (Video bank 1 starts at 16384, and the $8 in the high nybble of the VMCSB register sets color memory 8×1024 bytes higher than the base.)

Only bit 3 of the low nybble of VMCSB (or VM2 on the 128) is significant in bitmap mode. This bit is the 8K offset to the bitmap screen within the current video bank. It tells the computer whether the bitmap screen is to be located in the first half of the video bank (if set to zero) or in the second half (if set to one). And in this case, since we're placing the screen in the first half, starting at 16384, bit 3 is cleared.

After establishing the offset address of the high-resolution graphics screen and its color memory within the video bank, we actually assign a video bank number of 1 (defined as BNKNUM) using **VIDBNK**. Then we enter bitmap mode with **BITMAP**. On the 128, in this routine be sure to replace SCROLY at 53265 with its shadow register GRAPHM at 216. (See **BITMAP** for details on why this is done.)

After this, the high-resolution screen we've created is cleared with **CLRHRS**, a method employing self-modifying code which fills the screen with zeros. See **CLRHRS** for an explanation. (Using **CLRHRF** is another option.)

On the 128, just before clearing the screen, you can insert an STA MMUREG+1. This instruction causes the computer to

be placed into bank 0 as long as the accumulator contains a nonzero value. And, of course, this is where our bitmap resides on the 128. (Recall that above 16383, bank 15 is ROM.) So, if you PEEK the high-resolution screen here, you'll see its contents, rather than ROM, in bank 15.

At this point in the program, we use the current routine, **HRCOLF**, to fill color memory with bytes representing medium gray on black. Each byte of color memory, assigned to an 8 × 8 group of pixels on the bitmap, contains the foreground color value for these pixels in its high nybble and their background color value in the low nybble. The relative color values, defined as FORECL and BACKCL in the equates, are passed to the routine in .A and .X. (See **COLFIL** for a table of colors and their corresponding values.)

HRCOLF combines the two color values into a single byte and fills color memory with this byte within HRCLOP. The code for this memory-filling loop is similar to that used elsewhere in this book, and no description is needed here (see **CLRFIL** and **COLFIL**).

After color memory is filled, the program awaits a keypress before returning you to the normal text screen. The Kernal routine GETIN is used to fetch this keypress.

Since the Kernal is not present in bank 0, 128 users must switch to a bank where the Kernal is available. Here, we switch to bank 15 by storing a zero into the MMU configuration register at 65280. On a 128, add LDA #0:STA MMUREG to the code before calling GETIN.

When a key is pressed, **BITMAP** disables bitmap mode, and **VIDBNK** puts you back into the original 16K video bank (assumed to be bank 0, defined as BNKNM0). Commodore 128 users should see the normal text screen almost immediately as VM1 is copied to VMCSB on the next IRQ interrupt. But 64 users must physically reset the VIC-II chip memory control register before it becomes visible.

Routine

```
C000            ZP      =   251
C000            GETIN   =   65508
C000            VMCSB   =   53272    ; VIC-II chip memory control register
C000            SCROLY  =   53265    ; scroll/control register—use GRAPHM =
                                     ; 216 on the 128
C000            CI2PRA  =   56576    ; CIA 2 data port register A
C000            C2DDRA  =   56578    ; CIA 2 data direction register A
C000            FORECL  =   12       ; for medium gray foreground
C000            BACKCL  =   0        ; for black background
C000            SCREEN  =   24576    ; start of hi-res color memory
```

```
C000                MMUREG  =    65280        ; MMU configuration register (128 only)
C000                VM2     =    2605         ; VIC II chip memory control shadow register
                                              ; (128 only)
                                              ;
                                              ; Locate hi-res screen at 16384 and clear it,
                                              ; color memory (gray on black)
                                              ; at 24576. Enable/disable bitmap mode with
                                              ; BITMAP, clear hi-res screen
                                              ; with CLRHRS, and fill color memory with
                                              ; HRCOLF
C000  AD 18 D0              LDA  VMCSB        ; temporarily save VMCSB (64 only)
C003  8D 8F C0              STA  TEMP         ; (64 only)
                                              ; Now offset bitmap by 0K in video bank,
                                              ; locating color at 24K.
C006  A9 80                 LDA  #%10000000   ; LDA #%0xxx1000 if hi-res screen is in
                                              ; second half of video bank
C008  8D 18 D0              STA  VMCSB        ; reset register (replace VMCSB with VM2 on
                                              ; the 128)
                                              ; Now choose bank number.
C00B  AD 91 C0              LDA  BNKNUM       ; .A contains bank 0-3
C00E  20 76 C0              JSR  VIDBNK       ; select video bank 1
C011  20 6D C0              JSR  BITMAP       ; enter bitmap mode
                                              ; STA MMUREG+1; set the 128 to bank 0
                                              ; (128 only)
C014  20 33 C0              JSR  CLRHRS       ; clear the hi-res screen
C017  A9 0C                 LDA  #FORECL      ; foreground color for hi-res screen
C019  A2 00                 LDX  #BACKCL      ; and background color
C01B  20 51 C0              JSR  HRCOLF       ; clear hi res color memory
                                              ; LDA #0; set the 128 to bank 15 (128 only)
                                              ; STA MMUREG; (128 only)
C01E  20 E4 FF  WAIT        JSR  GETIN        ; get a keypress
C021  F0 FB                 BEQ  WAIT         ; if no keypress, then wait
C023  20 6D C0              JSR  BITMAP       ; turn off bitmap mode
C026  AD 92 C0              LDA  BNKNM0       ; return to original video bank; .A contains
                                              ; bank 0-3
C029  20 76 C0              JSR  VIDBNK       ; select bank 0
                                              ;
                                              ; Reset pointer to character set.
C02C  AD 8F C0              LDA  TEMP         ; (64 only)
C02F  8D 18 D0              STA  VMCSB        ; (64 only)
C032  60                    RTS
                                              ;
                                              ; Clear the hi-res screen with a self-
                                              ; modifying code method.
C033  AD 8E C0  CLRHRS      LDA  HRSCRN+1     ; store hi-res screen address in dummy
                                              ; location—$FFFF
C036  8D 46 C0              STA  LOOP+2
C039  AD 8D C0              LDA  HRSCRN
C03C  8D 45 C0              STA  LOOP+1
                                              ; Fill 32 pages with zeros.
C03F  A9 00                 LDA  #0
C041  A8                    TAY
C042  A2 20                 LDX  #32          ; 32 pages
C044  99 FF FF  LOOP        STA  $FFFF,Y      ; fill a block of 256 bytes with zeros
C047  C8                    INY
C048  D0 FA                 BNE  LOOP
C04A  EE 46 C0              INC  LOOP+2       ; page filled, so increase high-byte pointer
C04D  CA                    DEX
C04E  D0 F4                 BNE  LOOP         ; to fill all pages
C050  60                    RTS
                                              ;
                                              ; Clear hi-res color memory to FORECL on
                                              ; BACKCL.
```

```
C051  8E 90 C0  HRCOLF   STX    TEMPX          ; store BACKCL in .X temporarily
C054  0A                 ASL                   ; shift low nybble of FORECL into high
                                               ; nybble
C055  0A                 ASL
C056  0A                 ASL
C057  0A                 ASL
C058  0D 90 C0           ORA    TEMPX          ; .A now contains foreground color in high
                                               ; nybble, background in low nybble
C05B  A0 FA              LDY    #250
C05D  88        HRCLOP   DEY
C05E  99 00 60           STA    SCREEN,Y       ; first quarter
C061  99 FA 60           STA    SCREEN+250,Y
                                               ; second quarter
C064  99 F4 61           STA    SCREEN+500,Y
                                               ; third quarter
C067  99 EE 62           STA    SCREEN+750,Y
                                               ; fourth quarter
C06A  D0 F1              BNE    HRCLOP         ; fill all 250 bytes with color byte
C06C  60                 RTS
                                               ;
                                               ; Enable/disable bitmap mode.
C06D  AD 11 D0  BITMAP   LDA    SCROLY         ; substitute GRAPHM for SCROLY on the
                                               ; 128
C070  49 20              EOR    #%00100000     ; flip bit 5
C072  8D 11 D0           STA    SCROLY         ; reset register (again use GRAPHM instead
                                               ; of SCROLY on the 128)
C075  60                 RTS
                                               ;
                                               ; Select a 16K video bank. .A comes in
                                               ; containing the chosen bank number.
C076  49 03     VIDBNK   EOR    #3             ; effectively (3 - bank number)
C078  85 FB              STA    ZP             ; store it temporarily
C07A  AD 02 DD           LDA    C2DDRA         ; set data direction register for output
C07D  09 03              ORA    #3
C07F  8D 02 DD           STA    C2DDRA
                                               ;
C082  AD 00 DD           LDA    CI2PRA         ; take current CI2PRA value
C085  29 FC              AND    #252           ; and keep bits 2-7
C087  05 FB              ORA    ZP             ; OR with (3 - bank number)
C089  8D 00 DD           STA    CI2PRA         ; reset register
C08C  60                 RTS
                                               ;
C08D  00 40     HRSCRN   .WORD  16384          ; locate hi-res screen
C08F  00        TEMP     .BYTE  0              ; temporary storage for VMCSB configuration
C090  00        TEMPX    .BYTE  0              ; temporary storage for .X
C091  01        BNKNUM   .BYTE  1              ; bank 1
C092  00        BNKNM0   .BYTE  0              ; bank 0 or original bank
```

See also BITMAP, CLRHRF, CLRHRS, HRPOLR, HRSETP, PAINT.

Name

Set or clear a point on the hi-res screen based on polar coordinates

Description

Polar coordinates use two numbers, an angle and a distance value, to describe a position on a (usually circular) grid. **HRPOLR** translates these two numbers into a point on the hi-res screen and turns the point on or off.

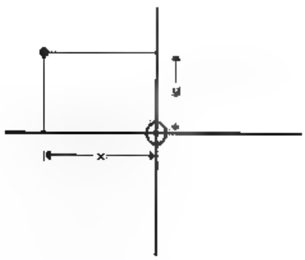
Prototype

1. Before beginning, create two lookup tables—one for 64 sine values, the other for 64 cosines (details below).
2. Start by looking up the sine (or cosine), based on the quadrant (0–3). Although this is a number in the range 0–255, it is treated as if it had a leading decimal point.
3. Multiply by LENGTH to find the *x* coordinate.
4. Add or subtract from the origin XORG and save the number.
5. Repeat the steps above, substituting cosine (or sine) to find the *y* coordinate.
6. Plot the resulting point on the hi-res screen.

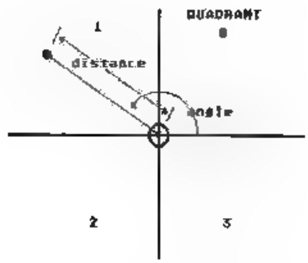
Explanation

To locate a point on a one-dimensional line, you need a single number representing the distance from the origin. On a two-dimensional flat plane, such as the hi-res screen, you need two numbers. The most common way to describe a point is to use the orthogonal Cartesian coordinate system (named for the French mathematician and philosopher René Descartes), which has two axes and *x* and *y* coordinates. A second, equally valid, method for plotting a point is to use polar coordinates, where the two numbers are an angle and a distance value. In the figure, the same point can be described in either Cartesian or polar terms.

Describing a Point



Cartesian Graph



Polar Graph

Angles are customarily measured in degrees (360 per circle) or radians (2π per circle). Both systems are rather arbitrary, and neither is especially well-suited to machine language. So a third method has been employed in the example routine, one that uses 256 "slices" per circle. Call these MLDegrees. Note that a right angle, which is 90 degrees or 0.5 π radians, is 64 MLDegrees. The advantage of this system is that an angle can be described by single byte. Also, by examining the two highest bits of the angle value, you can tell which quadrant the angle inhabits.

The **HRSETP** subroutine, which calculates and turns on a pixel, is described elsewhere in this book. The **MUL16** subroutine is basically the same as the **MULSHF** routine.

Before calling this routine, you have to create two lookup tables for the sine and cosine tables. Use the following short BASIC program:

```
10 FOR J=0 TO 63: RAD=J*(π/128): C=INT(COS
   (RAD)*256): S=INT(SIN(RAD)*256)
20 POKE 52992+J,C: POKE 53056+J,S: NEXT: END
```

This creates the cosine value table at 52992–53055 and the sine value table at 53056–53119. You can also include these values as a series of .BYTE statements, or they can be loaded from a disk file.

The two example routines draw a circle and a spiral. The circle routine keeps the length constant while stepping through the angles from 0 through 255 slices. The spiral pro-

gram does the same thing, but the length gradually decreases as the program runs.

Note: Before using this routine on the 128, enter POKE 216,255 or add the appropriate LDA and STA to the beginning of the program. Also, in the BASIC setup routine, substitute location 4864 for 52992 and location 4928 for 53056. These two locations, used for the table of sines and cosines, should be changed in the equates as well.

Routine

```
C000                Z1       =     251         ; pointer to the particular byte to be changed
C000                HRSCRN   =     $2000       ; screen is at 8192 decimal
C000                HRCOLR   =     $0400       ; color memory at 1024
C000                GETIN    =     $FFE4
C000                COSINE   =     52992       ; address of cosine value table
C000                SINE     =     53056       ; address of sine value table
                                               ;
C000  20 64 C1               JSR   HRSETUP     ; set up and clear the hi-res screen and color
                                               ; memory
C003  20 12 C0               JSR   CIRCLE      ; plot a circle
C006  20 2D C0               JSR   SPIRAL      ; and a spiral
C009  20 E4 FF      LOOPG    JSR   GETIN       ; wait
C00C  F0 FB                  BEQ   LOOPG
C00E  20 9F C1               JSR   HRCLEAR     ; turn off hi-res screen and restore to normal
C011  60                     RTS
                                               ;
C012  A9 00         CIRCLE   LDA   #0
C014  8D CA C0               STA   ANGLE       ; start at angle of 0
C017  A9 63                  LDA   #99         ; length of 99
C019  8D CB C0               STA   LENGTH
C01C  20 24 C0               JSR   CIRLP       ; down below
C01F  A9 32                  LDA   #50         ; second circle, radius of 50
C021  8D CB C0               STA   LENGTH
C024  20 4A C0      CIRLP    JSR   HRPOLR
C027  EE CA C0               INC   ANGLE
C02A  D0 F8                  BNE   CIRLP
C02C  60                     RTS
                                               ;
C02D  A9 00         SPIRAL   LDA   #0
C02F  8D CA C0               STA   ANGLE       ; angle starts at 0
C032  A9 64                  LDA   #100
C034  8D CB C0               STA   LENGTH      ; length is 100
C037  20 4A C0      SPLOOP   JSR   HRPOLR      ; plot it
C03A  EE CA C0               INC   ANGLE       ; add 1 to the angle
C03D  AD CA C0               LDA   ANGLE
C040  29 0F                  AND   #15         ; every 16 slices, the length decreases by 1
C042  D0 F3                  BNE   SPLOOP      ; not equal, loop back
C044  CE CB C0               DEC   LENGTH      ; length minus 1
C047  D0 EE                  BNE   SPLOOP      ; and loop back until 0
C049  60                     RTS               ; done
                                               ;
C04A  AD CA C0      HRPOLR   LDA   ANGLE       ; find the angle
C04D  29 3F                  AND   #$3F        ; strip off bits 6 and 7
C04F  AA                     TAX               ; look up
C050  BD 00 CF               LDA   COSINE,X    ; the cosine (0-255) from a table
C053  2C CA C0               BIT   ANGLE       ; check for quad 1 and 3
C056  50 03                  BVC   XXX         ; OK if 0 or 2
C058  BD 40 CF               LDA   SINE,X      ; else, load the sine
C05B  8D DD C1      XXX      STA   B1          ; get ready to multiply
```

```
C05E  AD CB C0                 LDA   LENGTH
C061  8D DE C1                 STA   B2             ; length in byte 2
C064  20 AF C1                 JSR   MUL16          ; multiply them
C067  AD CA C0                 LDA   ANGLE          ; check quadrant
C06A  29 C0                    AND   #%11000000     ; bits 6 and 7 are important
C06C  F0 11                    BEQ   PLUSX          ; two zeros
C06E  C9 C0                    CMP   #$C0           ; or two ones
C070  F0 0D                    BEQ   PLUSX          , mean add to XORG
C072  AD CC C0                 LDA   XORG           ; else, subtract
C075  38                       SEC
C076  ED E0 C1                 SBC   TM+1           ; the high byte
C079  8D CE C0                 STA   REALX          ; and save it
C07C  4C 89 C0                 JMP   CHECKY         ; now do the y location
C07F  AD CC C0  PLUSX          LDA   XORG           ; quadrant 0 or 3
C082  18                       CLC
C083  6D E0 C1                 ADC   TM+1           ; add the high byte
C086  8D CE C0                 STA   REALX          ; and store it
                                                    ;
C089  AD CA C0  CHECKY         LDA   ANGLE          ; get the angle again
C08C  29 3F                    AND   #$3F           ; bits 0-5
C08E  AA                       TAX
C08F  BD 40 CF                 LDA   SINE,X         ; get the sine
C092  2C CA C0                 BIT   ANGLE          ; check the quadrant
C095  50 03                    BVC   YYY
C097  BD 00 CF                 LDA   COSINE,X       ; else, get the cosine
C09A  8D DD C1  YYY            STA   B1             ; store it for multiplying
C09D  AD CB C0                 LDA   LENGTH         ; the length
C0A0  8D DE C1                 STA   B2             ; also
C0A3  20 AF C1                 JSR   MUL16          ; multiply them
C0A6  AD CD C0                 LDA   YORG           ; get y origin
C0A9  2C CA C0                 BIT   ANGLE          ; test the angle
C0AC  10 0A                    BPL   SUBTRACT       ; 128-255 mean subtract
C0AE  18                       CLC
C0AF  6D E0 C1                 ADC   TM+1           ; add the high byte
C0B2  8D CF C0                 STA   REALY          ; and store
C0B5  4C BF C0                 JMP   FORWD          ; skip the subtracting
C0B8  38        SUBTRACT       SEC
C0B9  ED E0 C1                 SBC   TM+1           ; subtract from YORG
C0BC  8D CF C0                 STA   REALY
                                                    ;
C0BF  AE CE C0  FORWD          LDX   REALX          ; get the point's x
C0C2  AC CF C0                 LDY   REALY          ; and y positions
C0C5  18                       CLC
C0C6  20 D0 C0                 JSR   HRSETP         ; and turn on the point
C0C9  60                       RTS
                                                    ;
C0CA  00        ANGLE          .BYTE 0
C0CB  00        LENGTH         .BYTE 0
C0CC  64        XORG           .BYTE 100            ; the center of the plotting area
C0CD  64        YORG           .BYTE 100            ; same for y
C0CE  00        REALX          .BYTE 0
C0CF  00        REALY          .BYTE 0
                                                    ;
C0D0            HRSETP         =     *              ; set a point on the hi-res screen
                                                    ; based on values in .X, .Y, and the carry flag
C0D0  20 3D C1                 JSR   SVREGS         ; save the registers
C0D3  20 DD C0                 JSR   HRCALC         ; calculate the location (in Z1) and the bit
                                                    ; pattern (MASK)
C0D6  20 27 C1                 JSR   POINT1         ; (subsitute POINT0 for turning off a pixel)
C0D9  20 50 C1                 JSR   LDREGS         ; restore the registers
C0DC  60                       RTS
                                                    ;
C0DD  08        HRCALC         PHP                  ; save the status register
```

```
C0DE  A9 00               LDA  #<HRSCRN     ; initialize Z1
C0E0  85 FB               STA  Z1           ; to point to
C0E2  A9 20               LDA  #>HRSCRN     ; the hi-res screen
C0E4  85 FC               STA  Z1+1
C0E6  98                  TYA               ; handle the row
C0E7  29 07               AND  #7           ; mask out the three low bits
C0E9  05 FB               ORA  Z1           ; and add them to Z1
C0EB  85 FB               STA  Z1
C0ED  98                  TYA               ; get .Y again
C0EE  4A                  LSR               ; shift right
C0EF  4A                  LSR               ; three
C0F0  4A                  LSR               ; times
C0F1  A8                  TAY               ; now .Y is a counter for adding 320
C0F2  F0 10               BEQ  ROWEND       ; if 0, skip the next part
C0F4  A9 40      ROWLP    LDA  #<320        ; low byte of 320
C0F6  18                  CLC
C0F7  65 FB               ADC  Z1           ; add to Z1
C0F9  85 FB               STA  Z1           ; store it
C0FB  A9 01               LDA  #>320        ; high byte
C0FD  65 FC               ADC  Z1+1         ; add to Z1
C0FF  85 FC               STA  Z1+1
C101  88                  DEY               ; loop back
C102  D0 F0               BNE  ROWLP
                                            ; Z1 now points to the left edge of the hi-res
                                            ; screen (1 of 200 rows).
                                            ;
C104  28         ROWEND   PLP               ; retrieve the carry flag
C105  90 02               BCC  TIMEX        ; if clear, the left side of the seam
C107  E6 FC               INC  Z1+1         ; otherwise, add 256 to the pointer
C109  8A         TIMEX    TXA               ; now do .X, the column
C10A  29 F8               AND  #%11111000   ; mask off 0-7 (the individual bits)
C10C  18                  CLC
C10D  65 FB               ADC  Z1           ; add to Z1
C10F  85 FB               STA  Z1           ; store it
C111  90 02               BCC  NOMORE       ; if carry is clear,
C113  E6 FC               INC  Z1+1         ; skip this INC
C115  A9 80      NOMORE   LDA  #$80         ; now set up mask
C117  8D 63 C1            STA  MASK
C11A  8A                  TXA               ; return .X to .A
C11B  29 07               AND  #%00000111   ; bottom three bits (0-7 value)
C11D  F0 07               BEQ  CLOSEUP      ; if zero, skip it
C11F  AA                  TAX               ; otherwise, set up .X for a counter
C120  4E 63 C1   XLOOP    LSR  MASK         ; move it right
C123  CA                  DEX               ; count down
C124  D0 FA               BNE  XLOOP
C126  60         CLOSEUP  RTS               ; Finished. Z1 points to the byte and MASK
                                            ; holds the bitmask.
                                            ;
C127  A0 00      POINT1   LDY  #0           ; this sets a point on the screen
C129  AD 63 C1            LDA  MASK         ; get the mask
C12C  11 FB               ORA  (Z1),Y       ; turn on a pixel
C12E  91 FB               STA  (Z1),Y       ; put it on the screen
C130  60                  RTS               ; and that's all
                                            ;
C131  A0 00      POINT0   LDY  #0           ; almost the same as POINT1, but it clears a
                                            ; pixel
C133  AD 63 C1            LDA  MASK         ; get the bitmask
C136  49 FF               EOR  #$FF         ; flip the bits
C138  31 FB               AND  (Z1),Y       ; AND instead of OR
C13A  91 FB               STA  (Z1),Y       ; store it
C13C  60                  RTS               ; finished
                                            ;
C13D  08         SVREGS   PHP               ; first save .P
C13E  48                  PHA               ; then .A
```

```
C13F  08                   PHP                ; then .P again
C140  8D  5F  C1           STA   TEMPA        ; save .A
C143  8E  60  C1           STX   TEMPX        ; .X
C146  8C  61  C1           STY   TEMPY        ; and .Y
C149  68                   PLA                ; pull .P into .A
C14A  8D  62  C1           STA   TEMPP        ; store it
C14D  68                   PLA                ; get .A again
C14E  28                   PLP                ; and .P
C14F  60                   RTS
                                              ;
C150  AE  60  C1  LDREGS   LDX   TEMPX        ; restore .X
C153  AC  61  C1           LDY   TEMPY        ; and .Y
C156  AD  62  C1           LDA   TEMPP        ; get .P
C159  48                   PHA                ; push it
C15A  AD  5F  C1           LDA   TEMPA        ; get .A back
C15D  28                   PLP                ; and restore .P
C15E  60                   RTS                ; done
                                              ;
C15F  00          TEMPA    .BYTE 0
C160  00          TEMPX    .BYTE 0
C161  00          TEMPY    .BYTE 0
C162  00          TEMPP    .BYTE 0
C163  00          MASK     .BYTE 0
                                              ;
C164  A9  3B      HRSETUP  LDA   #59          ; to set up the hi-res screen at $2000
C166  8D  11  D0           STA   53265        ; put a 59 into 53265
C169  A9  18               LDA   #24
C16B  8D  18  D0           STA   53272        ; and a 24 into 53272
C16E  A9  10               LDA   #$10         ; white and black
C170  A0  00               LDY   #0           ; index into color memory
C172  99  00  04  COLLP    STA   HRCOLR,Y
C175  99  FA  04           STA   HRCOLR+250,Y
C178  99  F4  05           STA   HRCOLR+500,Y
C17B  99  EE  06           STA   HRCOLR+750,Y
C17E  C8                   INY
C17F  C0  FA               CPY   #250         ; fill 1000 bytes
C181  D0  EF               BNE   COLLP
                                              ;
C183  A9  00               LDA   #<HRSCRN     ; now set up the clear screen routine
C185  8D  93  C1           STA   FAKE+1
C188  A9  20               LDA   #>HRSCRN     ; high byte
C18A  8D  94  C1           STA   FAKE+2
C18D  A2  20               LDX   #32          ; 32 pages
C18F  A0  00               LDY   #0
C191  98                   TYA                ; zero for cleared bits
C192  99  FF  FF  FAKE     STA   $FFFF,Y
C195  C8                   INY
C196  D0  FA               BNE   FAKE
C198  EE  94  C1           INC   FAKE+2       ; increment the high byte
C19B  CA                   DEX
C19C  D0  F4               BNE   FAKE
C19E  60                   RTS
                                              ;
C19F  A9  1B      HRCLEAR  LDA   #27          ; turn off hi-res
C1A1  8D  11  D0           STA   53265        ; 27 into 53265
C1A4  A9  15               LDA   #21
C1A6  8D  18  D0           STA   53272        ; 21 into 53272
C1A9  A9  93               LDA   #147         ; clear screen
C1AB  20  D2  FF           JSR   $FFD2
C1AE  60                   RTS
C1AF              MUL16    =     *            ; multiplies two numbers
C1AF  A9  00               LDA   #0           ; zero out
C1B1  8D  DF  C1           STA   TM           ; low byte
```

```
C1B4  8D E0 C1            STA    TM+1      ; and high byte of the result
C1B7  A2 08               LDX    #8        ; eight cycles
C1B9  AD DD C1  MULSTR    LDA    B1
C1BC  2E DE C1            ROL    B2        ; multiply or not?
C1BF  90 0F               BCC    NOMULT    ; no, it's a zero
C1C1  18                  CLC
C1C2  6D DF C1            ADC    TM        ; add B1 to TM
C1C5  8D DF C1            STA    TM        ; store it
C1C8  A9 00               LDA    #0        ; and the
C1CA  6D E0 C1            ADC    TM+1      ; high byte
C1CD  8D E0 C1            STA    TM+1      ; in TM
C1D0  CA        NOMULT    DEX              ; count down (eight bits)
C1D1  D0 01               BNE    MLMORE    ; not equal yet
C1D3  60                  RTS              ; the main return of MUL16
C1D4  0E DF C1  MLMORE    ASL    TM        ; move it left
C1D7  2E E0 C1            ROL    TM+1      ; and the high byte
C1DA  4C B9 C1            JMP    MULSTR    ; go back
                                           ;
C1DD  00        B1        .BYTE  0
C1DE  00        B2        .BYTE  0
C1DF  00 00     TM        .BYTE  0,0
```

See also BITMAP, CLRHRF, CLRHRS, HRCOLF, HRSETP, PAINT.

Name

Set or clear a point on the hi-res screen

Description

Enter this routine with the *x* coordinate of the point in the .X register and carry flag and the *y* coordinate (0–199) in the .Y register. The corresponding point on the hi-res screen is then turned on. Because of the unusual way that hi-res memory is laid out, most of the routine is devoted to shuffling numbers around, calculating the appropriate memory location.

Prototype

1. Save the register values.
2. Calculate the memory location by first setting a zero page location to point to the start of hi-res screen memory.
3. Next, add in the lower three bits of .Y (0–7).
4. Divide .Y by 8, and add 320 that number of times.
5. Mask off the lower three bits of .X and add the result.
6. Use the lower three bits as a counter to rotate the bit to its proper place in the MASK variable.
7. Set the point by putting a zero in .Y, MASK in .A, and ORA indirectly off the zero-page pointer.
8. To clear the point, exclusive-OR MASK with $FF and AND it with the memory location.
9. Restore the original register values.

Explanation

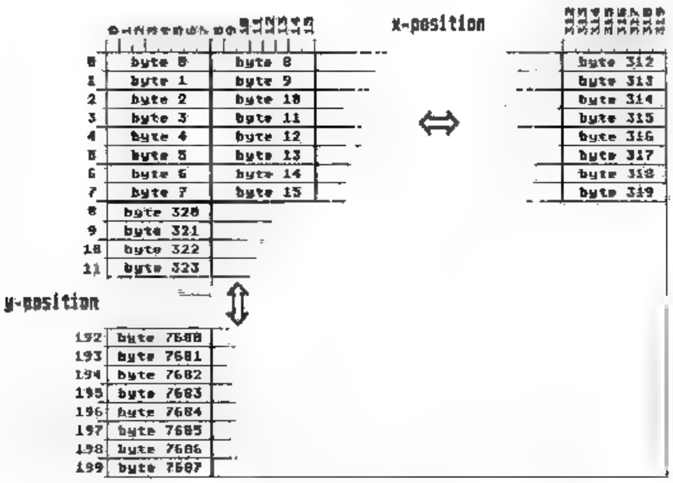
The horizontal width of the hi-res screen is 320 pixels (numbered 0–319). The vertical height is 200 lines (0–199). The total of 64,000 points fit into exactly 8000 bytes, because each byte has eight bits that control eight screen pixels. Hi-res screen memory is laid out in a manner very similar to the text screen.

This up and down zig-zagging pattern causes a few difficulties. The HRCALC subroutine at $C02F–$C078 must go through some contortions to figure out just where a given point is located in memory. Initially, the starting location of the hi-res screen (8192, in the example) is stored in the zero-page pointer Z1 ($FB–$FC).

The *y* position is handled first. It has two components: bits 0–2 and bits 3–7. Bits 0–2 hold a value between 0 and 7 that can be added directly to the Z1 pointer. Bits 3–7 hold values divisible by 8 (0, 8, 16, 24, and so on). Each time the value in *y* increases by 8, the screen memory increases by 320

(see figure). Starting at $C040, the value in .Y is divided by 8, and a loop adds 320 to Z1 as many times as is needed.

Hi-Res Screen Organization



The X register is limited to holding a number from 0 through 255, but the *x* coordinates run from 0 through 319. The carry flag is used as an extension of the X register. If the point is higher than 255, set the carry flag and load .X with the coordinate of the point minus 256. If it's 0–255, carry should be clear. The carry flag setting must be saved at the start of HRCALC, where the processor flags are pushed on the stack with PHP. At $C056, PLP restores the flags, including carry. If carry is set, the high byte of Z1 is increased by one.

Like the *y* position, the *x* position must be divided into two parts—the first three bits and the last five bits. Note that in the top row, *x* coordinates 0–7 fit into byte 0, 8–15 fit into byte 8, and so on. If the bottom three bits are cleared, the result can be added to Z1 to pinpoint the memory location to be changed.

All that remains is to take the number %10000000 and rotate it to the right to get the single 1 bit into the correct position. The lower three bits of .X are used in a loop that rotates MASK to the right.

When HRCALC is finished with its calculations, the memory address is in $FB–$FC, and the mask value is in MASK. Now either POINT1 or POINT0 can be called to turn the pixel on or off.

The framing routine at the very beginning starts .X at 0 and .Y at 150, and draws a diagonal line from the bottom left corner to the top right. HRSETUP and HRCLEAR enter and exit hi-res mode. Note that no ROM routines are called, except for GETIN, which waits for a key to be pressed before exiting to BASIC.

Note: Before using this routine on the 128, enter POKE 216,255 (or add the line LDA #255: STA 216 to the program).

Routine

```
C000                Z1       =    251        ; pointer to the particular byte to be changed
C000                HRSCRN   =    $2000      ; screen is at 8192 decimal
C000                HRCOLR   =    $0400      ; color memory at 1024
C000                GETIN    =    $FFE4
                                             ;
C000  20  B6  C0             JSR  HRSETUP    ; set up and clear the hi-res screen and color
                                             ; memory
C003  A2  00                 LDX  #0
C005  A0  96                 LDY  #150
C007  18                     CLC
C008  20  22  C0    MAIN     JSR  HRSETP     ; turn on the point
C00B  E8                     INX             ; and its neighbor
C00C  D0  01                 BNE  NSET       ; if not zero, continue
C00E  38                     SEC             ; else, set carry for the seam
C00F  20  22  C0    NSET     JSR  HRSETP     ; next one
C012  E8                     INX
C013  D0  01                 BNE  NSEU       ; handle the overflow
C015  38                     SEC
C016  88            NSEU     DEY
C017  D0  EF                 BNE  MAIN
C019  20  E4  FF    GL       JSR  GETIN      ; get a key
C01C  F0  FB                 BEQ  GL         ; wait before exiting
                                             ;
C01E  20  F1  C0             JSR  HRCLEAR    ; turn off hi-res screen and restore to normal
C021  60                     RTS
                                             ;
C022                HRSETP   =    *          ; set a point on the hi-res screen
                                             ; based on values in .X, .Y, and the carry
                                             ; flag
C022  20  8F  C0             JSR  SVREGS     ; save the registers
C025  20  2F  C0             JSR  HRCALC     ; calculate the location (in Z1) and the bit
                                             ; pattern (MASK)
C028  20  79  C0             JSR  POINT1     ; (substitute POINT0 for turning off a pixel)
C02B  20  A2  C0             JSR  LDREGS     ; restore the registers
C02E  60                     RTS
                                             ;
C02F  08            HRCALC   PHP             ; save the status register
C030  A9  00                 LDA  #<HRSCRN   ; initialize Z1
C032  85  FB                 STA  Z1         ; to point to
C034  A9  20                 LDA  #>HRSCRN   ; the hi-res screen
C036  85  FC                 STA  Z1+1
                                             ;
C038  98                     TYA             ; handle the row
C039  29  07                 AND  #7         ; mask out the three low bits
C03B  05  FB                 ORA  Z1         ; and add them to Z1
C03D  85  FB                 STA  Z1
C03F  98                     TYA             ; get .Y again
C040  4A                     LSR             ; shift right
C041  4A                     LSR             ; three
C042  4A                     LSR             ; times
```

```
C043  A8                 TAY                    ; now .Y is a counter for adding 320
C044  F0 10              BEQ   ROWEND           ; if zero, skip the next part
C046  A9 40     ROWLP    LDA   #<320            ; low byte of 320
C048  18                 CLC
C049  65 FB              ADC   Z1               ; add to Z1
C04B  85 FB              STA   Z1               ; store it
C04D  A9 01              LDA   #>320            ; high byte
C04F  65 FC              ADC   Z1+1             ; add to Z1
C051  85 FC              STA   Z1+1
C053  88                 DEY                    ; loop back
C054  D0 F0              BNE   ROWLP
                                                ;
                                                ; Z1 now points to the left edge of the hi-
                                                ; res screen (1 of 200 rows).
                                                ;
C056  28        ROWEND   PLP                    ; retrieve the carry flag
C057  90 02              BCC   TIMEX            ; if clear, the left side of the seam
C059  E6 FC              INC   Z1+1             ; otherwise, add 256 to the pointer
C05B  8A        TIMEX    TXA                    ; now do .X, the column
C05C  29 F8              AND   #%11111000       ; mask off 0-7 (the individual bits)
C05E  18                 CLC
C05F  65 FB              ADC   Z1               ; add to Z1
C061  85 FB              STA   Z1               ; store it
C063  90 02              BCC   NOMORE           ; if carry's clear,
C065  E6 FC              INC   Z1+1             ; skip this INC
C067  A9 80     NOMORE   LDA   #$80             ; now set up MASK
C069  8D B5 C0           STA   MASK
C06C  8A                 TXA                    ; return .X to .A
C06D  29 07              AND   #%00000111       ; bottom three bits (0-7 value)
C06F  F0 07              BEQ   CLOSEUP          ; if zero, skip it
C071  AA                 TAX                    ; otherwise, set up .X for a counter
C072  4E B5 C0  XLOOP    LSR   MASK             ; move it right
C075  CA                 DEX                    ; count down
C076  D0 FA              BNE   XLOOP
C078  60        CLOSEUP  RTS                    ; Finished. Z1 points to the byte and MASK
                                                ; holds the bitmask.
                                                ;
C079  A0 00     POINT1   LDY   #0               ; this sets a point on the screen
C07B  AD B5 C0           LDA   MASK             ; get the mask
C07E  11 FB              ORA   (Z1),Y           ; turn on a pixel
C080  91 FB              STA   (Z1),Y           ; put it on the screen
C082  60                 RTS                    ; and that's all
                                                ;
C083  A0 00     POINT0   LDY   #0               ; almost the same as POINT1, but it clears
                                                ; a pixel
C085  AD B5 C0           LDA   MASK             ; get the bit mask
C088  49 FF              EOR   #$FF             ; flip the bits
C08A  31 FB              AND   (Z1),Y           ; AND instead of OR
C08C  91 FB              STA   (Z1),Y           ; store it
C08E  60                 RTS                    ; finished
                                                ;
C08F  08        SVREGS   PHP                    ; first save .P
C090  48                 PHA                    ; then .A
C091  08                 PHP                    ; then .P again
C092  8D B1 C0           STA   TEMPA            ; save .A
C095  8E B2 C0           STX   TEMPX            ; .X
C098  8C B3 C0           STY   TEMPY            ; and .Y
C09B  68                 PLA                    ; pull .P into .A
C09C  8D B4 C0           STA   TEMPP            ; store it
C09F  68                 PLA                    ; get .A again
C0A0  28                 PLP                    ; and .P
C0A1  60                 RTS
                                                ;
```

```
C0A2  AE B2  C0  LDREGS   LDX   TEMPX         ; restore .X
C0A5  AC B3  C0           LDY   TEMPY         ; and .Y
C0A8  AD B4  C0           LDA   TEMPP         ; get .P
C0AB  48                  PHA                 ; push it
C0AC  AD B1  C0           LDA   TEMPA         ; get .A back
C0AF  28                  PLP                 ; and restore .P
C0B0  60                  RTS                 ; done
                                              ;
C0B1  00         TEMPA    .BYTE 0
C0B2  00         TEMPX    .BYTE 0
C0B3  00         TEMPY    .BYTE 0
C0B4  00         TEMPP    .BYTE 0
C0B5  00         MASK     .BYTE 0
                                              ;
C0B6  A9 3B      HRSETUP  LDA   #59           ; to set up the hi-res screen at $2000
C0B8  8D 11  D0           STA   53265         ; put a 59 into 53265
C0BB  A9 18               LDA   #24
C0BD  8D 18  D0           STA   53272         ; and a 24 into 53272
C0C0  A9 10               LDA   #$10          ; white and black
C0C2  A0 00               LDY   #0            ; index into color memory
C0C4  99 00  04  COLLP    STA   HRCOLR,Y
C0C7  99 FA  04           STA   HRCOLR+250,Y
C0CA  99 F4  05           STA   HRCOLR+500,Y
C0CD  99 EE  06           STA   HRCOLR+750,Y
C0D0  C8                  INY
C0D1  C0 FA               CPY   #250          ; fill 1000 bytes
C0D3  D0 EF               BNE   COLLP
                                              ;
C0D5  A9 00               LDA   #<HRSCRN      ; Now set up the clear-screen routine.
C0D7  8D E5  C0           STA   FAKE+1
C0DA  A9 20               LDA   #>HRSCRN      ; high byte
C0DC  8D E6  C0           STA   FAKE+2
C0DF  A2 20               LDX   #32           ; 32 pages
C0E1  A0 00               LDY   #0
C0E3  98                  TYA                 ; zero for cleared bits
C0E4  99 FF  FF  FAKE     STA   $FFFF,Y
C0E7  C8                  INY
C0E8  D0 FA               BNE   FAKE
C0EA  EE E6  C0           INC   FAKE+2        ; increment the high byte
C0ED  CA                  DEX
C0EE  D0 F4               BNE   FAKE
C0F0  60                  RTS
                                              ;
C0F1  A9 1B      HRCLEAR  LDA   #27           ; Turn off hi res.
C0F3  8D 11  D0           STA   53265         ; 27 into 53265
C0F6  A9 15               LDA   #21
C0F8  8D 18  D0           STA   53272         ; 21 into 53272
C0FB  A9 93               LDA   #147          ; clear screen
C0FD  20 D2  FF           JSR   $FFD2
C100  60                  RTS
```

See also BITMAP, CLRHRF, CLRHRS, HRCOLF, HRPOLR, PAINT.

Name

Increment a two-byte counter

Description

The machine language INC instruction increments a value in memory by one. This **INC2** routine extends the usefulness of INC to cover a wider range of values (0–65535 instead of 0–255).

Prototype

1. INCrement the low byte of a counter.
2. If it has reached zero, increment the high byte.
3. If the high byte has reached zero, the counter has gone past the limit of 65535. Set the carry flag to indicate an error.

Explanation

The example program waits for a keypress and exits if the F1 key is detected. Otherwise, it prints the character and calls **INC2** to keep track of how many keys have been pressed.

Within the **INC2** subroutine, the low byte of COUNTER is increased by one. If it reaches zero, the high byte is also increased. Then the carry flag is cleared, and the subroutine ends. Clearing the carry flag isn't necessary, but it's included to signal a successful two-byte increment. If **INC2** ever counts beyond the top limit ($FFFF), carry is set to indicate an overflow.

Back in the main routine, the program ends when F1 is pressed or if the user presses more than 65,535 keys. At that point, two RETURNs are printed followed by the number of keystrokes.

Note to 128 users: Since this program checks for the F1 key, which is predefined to print GRAPHIC, you should add the line KEY1, CHR$(133) to insure that the program works properly on the 128. Alternately, you could call the Kernal routine PFKEY at $FF65. This routine redefines a given function key.

Routine

```
C000                F1       =    133
C000                GETIN    =    $FFE4
C000                CHROUT   =    $FFD2
C000                LINPRT   =    $BDCD       ; LINPRT = $8E32 on the 128—ROM
                                              ; routine to print a number
C000  A9  00                 LDA  #0          ; clear the counter
C002  8D  41  C0             STA  COUNTER
C005  8D  42  C0             STA  COUNTER+1
                                              ;
```

```
C008  20  E4  FF  MLOOP    JSR   GETIN       ; get a keypress
C00B  F0  FB               BEQ   MLOOP       ; loop until it happens
C00D  C9  85               CMP   #F1         ; is it the F1 key?
C00F  F0  08               BEQ   CLEANUP     ; yes, finish up
C011  20  D2  FF           JSR   CHROUT      ; else, print it
C014  20  2B  C0           JSR   INC2        , and the counter clicks
C017  90  EF               BCC   MLOOP       ; carry clear means less than 65535 characters
                                             ; fall through to CLEANUP if carry set after
                                             , INC2
C019  A9  0D      CLEANUP  LDA   #13         ; RETURN character
C01B  20  D2  FF           JSR   CHROUT      ; print it
C01E  20  D2  FF           JSR   CHROUT      ; print it again
C021  AE  41  C0           LDX   COUNTER     , low byte of counter value
C024  AD  42  C0           LDA   COUNTER+1   ; high byte
C027  20  CD  BD           JSR   LINPRT      , print the number of keys pressed
C02A  60                   RTS
                                             ;
C02B  EE  41  C0  INC2     INC   COUNTER     ; add one to the counter
C02E  F0  [illegible]      BEQ   INCHI       ; if equal to zero, increment the high byte
C030  18          FINIS    CLC               ; clear carry (meaning OK)
C031  60                   RTS               ; and return
C032  EE  42  C0  INCHI    INC   COUNTER+    ; up the high byte
C035  D0  F9               BNE   FINIS       ; if it's not zero, OK
C037  A9  FF               LDA   #$FF
C039  [illegible]  41  C0  STA   COUNTER
C03C  8D  [illegible]  C0  STA   COUNTER+
C03F  [illegible]          SEC               ; carry set means we've reached the limit
C040  60                   RTS
                                             ,
C041  00  00      COUNTER  .BYTE 0,0
```

See also ADDBYT, ADDFP, ADDINT.

Name

Initialize a disk

Description

INITLZ initializes a disk, forcing the block allocation map (BAM) to be read into the disk drive's memory. This is sometimes useful after a new disk has been inserted or after changes have been made to the files on the disk.

Prototype

1. Open the disk command channel, channel 15.
2. As part of the filename, send the initialize command, I0.
3. Close the command channel.

Explanation

Brand-new blank disks must be formatted before they can be used. On some computers, this process is called *initializing* a disk. On Commodore computers, however, initializing has quite a different meaning.

When you send the DOS command *I0*, the disk drive reads the current block allocation map into memory, so it knows which sectors are already taken. This process should happen automatically when the disk drive senses that a new disk has been inserted. But it doesn't hurt to force an initialization. It may even be necessary if you tamper with file information (unscratching a file, for example).

The program works like most of the other DOS routines. It opens channel 15, the disk command channel, with the Kernal SETLFS routine. Then, in the process of setting the name, it uses the two characters *I0*. When the file is opened (with Kernal SETNAM and OPEN), the command is automatically sent to the drive. Then the file is closed and channels are cleared.

Routine

```
C000                SETLFS   =     $FFBA
C000                SETNAM   =     $FFBD
C000                OPEN     =     $FFC0
C000                CLOSE    =     $FFC3
C000                CLRCHN   =     $FFCC
                                                 ;
C000  A9 01         INITLZ   LDA   #1            ; logical file number
C002  A2 08                  LDX   [redacted]    ; device number for disk drive
C004  A0 0F                  LDY   #15           ; secondary address for command channel
C006  20 BA FF               JSR   SETLFS        ; prepare to open file
C009  A9 03                  LDA   #BUFLEN       ; length of buffer
C00B  A2 1E                  LDX   #<BUFFER      ; .X and .Y hold the
C00D  A0 C0                  LDY   #>BUFFER      ; address of the buffer
C00F  20 BD FF               JSR   SETNAM        ; set up filename
```

```
C012  20  C0  FF            JSR   OPEN        ; open it
C015  A9  01                LDA   #1          ; and immediately
C017  20  C3  FF            JSR   CLOSE       ; close the command channel
C01A  20  CC  FF            JSR   CLRCHN      ; clear the channels
C01D  60                    RTS               ; all done
                                              ; data area
C01E  49  30       BUFFER   .ASC  "I0"
C020  0D                    .BYTE 13          ; RETURN character
C021               BUFLEN   =     * - BUFFER
```

See also CONCAT, COPYFL, FORMAT, RENAME, SCRTCH, VALIDT.

Name

Interrupt-driven clock

Description

This routine updates a digital clock at the upper right corner of the screen during each IRQ interrupt. This clock relies on the first time-of-day clock (TOD 1) to maintain accurate time.

A feature of this routine is that it allows you to toggle the clock display on or off by pressing the F7 key. If the clock distracts you or becomes annoying, simply press F7 and clear the screen. (On the 128, before SYSing to the routine, you'll need to define the F7 key to a null string by entering KEY 7,"".)

To disable the clock altogether, press RUN/STOP–RESTORE to reset the IRQ interrupt vector.

Prototype

This is actually a two-part routine. Before entering the first part (**INTCLK**), store the current time in binary-coded decimal format as TIMSET at the end of the program. Be sure to add $80 to the hours byte if the time is p.m. (See **TOD2ST** for details on setting the time-of-day clocks.)

In **INTCLK**:
1. Using **TOD1ST**, set TOD 1 clock to the time specified in TIMSET.
2. Disable IRQ interrupts with SEI.
3. Redirect the IRQ interrupt vector at 788–789 to MAIN.
4. With the vector changed, reenable IRQ interrupts and RTS.

In MAIN:
1. Determine whether the last key pressed was F7. If it was, toggle a clock display flag from 0 to 1, or vice versa, with EOR #1.
2. If the clock display flag contains a zero, exit the routine through the normal IRQ interrupts (in step 7).
3. Otherwise, store the current cursor color (COLOR) into each color RAM position for the clock display. Then store the current screen background color in the initial color position.
4. In PLACLP, read and store to the clock display in reverse video the digits for the hour, minute, and second. Precede each digit pair with a reverse colon. (The first colon is not seen because its color is the screen background color.)
5. Print a reverse decimal and the tenths of seconds.

6. If the hours byte is negative, print a *P* for p.m.; otherwise, print an *A*.
7. Exit by executing the normal IRQ interrupts.

Explanation

The actual readout for the clock is stored to the screen during the routine MAIN. Within this routine, the Y register is used to index the screen position in the clock display, while .X points to the relative TOD clock bytes—either hours, minutes, seconds, or tenths of seconds.

First, MAIN fills the underlying color RAM for the display with the current cursor color (as stored in COLOR). This takes place in COLOOP. Because the clock is displayed in the current text color, the readout will be visible regardless of the screen background color (assuming, of course, that the text color differs from the screen background color).

After COLOOP, the clock itself is stored to the screen. Each digit pair within the clock—representing hours, minutes, and seconds—is separated by a reverse colon for better readability. A reverse decimal point is located between the seconds place and tenths-of-seconds place at $C05B.

Notice also that a colon is placed just before the clock display. This colon doesn't actually appear on the screen since its color byte is taken from the screen background color register. Nevertheless, it prevents the clock display from being accepted as a BASIC line if the user should accidentally hit RETURN over this line.

Bytes from the TOD clock are in binary-coded decimal format. The high nybble of each byte represents the ten's place, while the low nybble is the one's place. By alternately masking low and high nybbles and converting the result to screen codes in PLACLP, you can store each byte from the TOD clock reading in screen memory as a two-digit number. Since bit 7 is the a.m./p.m. flag in the hours byte, it must be masked in order to read the hours digits correctly.

The exception to this arrangement within the TOD clock is the tenths-of-seconds place. Since no more than a single decimal digit need be stored in the tenths byte, the high nybble is unused. As a result, we needn't break this byte into separate nybbles. We simply store it after converting it to a screen code.

The last thing to be done in the routine, before exiting to the normal IRQ interrupt handler, is to display the *A* or *P* for

a.m. or p.m. The code for this begins at $C068.

Note: **INTCLK** currently uses TOD1 (the clock in CIA #1) to keep time. If, for some reason, this clock is unavailable, you can just as easily use TOD2 by substituting TODTN2 for TODTN1 in the program.

Routine

```
C000                TODTN1   =     56328        ; time-of-day clock 1 -tenths-of-seconds
                                                ; register
C000                TODTN2   =     56584        ; time-of-day clock 2 tenths-of-seconds
                                                ; register
C000                IRQVEC   =     788          ; vector to IRQ interrupt routine
C000                IRQNOR   =     59953        ; IRQNOR = 64101 on 128—normal
                                                ; interrupt service routine
C000                LSTX     =     197          ; LSTX = 213 on the 128—last key pressed
C000                SCRCLK   =     1050         ; screen address for the clock
C000                COLCLK   =     55322        ; color RAM for clock
C000                BGCOL0   =     53281        ; background color register for screen
C000                COLOR    =     646          ; COLOR = 241 on the 128—text foreground
                                                ; color register
                                                ;
                                                ; Set up an interrupt-driven clock display.
                                                ; Replace TODTN1 with TODTN2 to use
                                                ; TOD clock 2.
C000 20 7F C0 INTCLK   JSR   TOD1ST       ; set TOD clock 1 and start it by writing to
                                                ; tenths
C003 78                SEI                ; disable IRQ interrupts to change the IRQ
                                                ; vector
C004 A9 10             LDA   #<MAIN       ; store the low byte of interrupt wedge
C006 8D 14 03          STA   IRQVEC
C009 A9 C0             LDA   #>MAIN       ; and the high byte
C00B 8D 15 03          STA   IRQVEC+1
C00E 58                CLI                ; reenable IRQ interrupts
C00F 60                RTS                ; exit setup routine
                                                ;
C010 A5 C5    MAIN     LDA   LSTX         ; check for F7
C012 C9 03             CMP   #3           ; is it F7?
C014 D0 08             BNE   NOTTOG       ; don't toggle the clock if not F7
C016 AD 92 C0 TOGGLE   LDA   CLKFLG       ; toggle clock on/off
C019 49 01             EOR   #1
C01B 8D 92 C0          STA   CLKFLG       ; reset flag
C01E AD 92 C0 NOTTOG   LDA   CLKFLG       ; necessary for NOTTOG
C021 F0 4F             BEQ   EXIT         ; if flag is zero, don't show the clock
                                                ; instead, execute normal IRQs
C023 A0 0B             LDY   #11          ; make clock color the same as text color
C025 AD 86 02          LDA   COLOR        ; get cursor color
C028 99 1A D8 COLOOP   STA   COLCLK,Y     ; store it to each color RAM position
C02B 88                DEY                ; next lower position
C02C D0 FA             BNE   COLOOP       ; do 12 positions
C02E AD 21 D0          LDA   BGCOL0       ; get background color for first colon
C031 8D 1A D8          STA   COLCLK       ; so first colon is not seen
C034 A2 03             LDX   #3           ; as an index for hrs., mins., secs., tenths
C036 A0 FF             LDY   #255         ; so .Y starts with zero in PLACLP
C038 C8       PLACLP   INY                ; for next position in the clock
C039 20 79 C0          JSR   COLON        ; POKE in colon at beginning of clock
C03C C8                INY                ; for next position
C03D BD 08 DC          LDA   TODTN1,X     ; start with hrs.
C040 48                PHA                ; store it temporarily
C041 29 70             AND   #%01110000   ; mask out low nybble and bit 7
C043 4A                LSR                ; shift high nybble into low nybble
C044 4A                LSR
```

```
C045  4A                    LSR
C046  4A                    LSR
C047  09 B0                 ORA   #176        ; convert to numeric range (+48), reverse
                                              ; (+128)
C049  99 1A 04              STA   SCRCLK,Y    ; position the result on the screen
C04C  C8                    INY               ; for next position
C04D  68                    PLA               ; retrieve byte to handle low nybble
C04E  29 0F                 AND   #$0F        ; mask out high nybble
C050  09 B0                 ORA   #176        ; convert to numeric range, reverse
C052  99 1A 04              STA   SCRCLK,Y    ; and store result to screen
C055  CA                    DEX               ; for next place—mins. and secs.
C056  D0 E0                 BNE   PLACLP      ; do three bytes—hrs., mins., secs.
C058  C8                    INY               ; to position decimal
C059  A9 AE                 LDA   #174        ; screen code for a reverse decimal
C05B  99 1A 04              STA   SCRCLK,Y    ; POKE it
C05E  C8                    INY               ; to position tenths place
C05F  AD 08 DC              LDA   TODTN1      ; get the tenths byte and restart the clock
C062  09 B0                 ORA   #176        ; convert to numeric range and reverse
C064  99 1A 04              STA   SCRCLK,Y    ; display the tenths
C067  C8                    INY               ; to position a.m./p.m.
C068  AD 0B DC              LDA   TODTN1+3    ; read hours
C06B  30 08                 BMI   PMFLAG      ; bit 7 is set indicating p.m. time
C06D  A9 81                 LDA   #129        ; screen code for reverse A—a.m.
C06F  99 1A 04  PRAMPM      STA   SCRCLK,Y    ; store it to screen
C072  4C 31 EA  EXIT        JMP   IRQNOR      ; exit always
C075  A9 90     PMFLAG      LDA   #144        ; screen code for P
C077  D0 F6                 BNE   PRAMPM      ; print P and exit to normal interrupts
                                              ;
                                              ; POKE in a reverse colon at current screen
                                              ; position.
C079  A9 BA     COLON       LDA   #186
C07B  99 1A 04              STA   SCRCLK,Y
C07E  60                    RTS
                                              ;
                                              ; Set TOD clock 1 (or 2).
                                              ; Replace TODTN1 with TODTN2 to set TOD
                                              ; clock 2.
C07F  A0 00     TOD1ST      LDY   #0          ; as an index for the time setting
C081  A2 03                 LDX   #3          ; as an index for hrs., mins., secs., tenths
C083  B9 8F C0  SETLOP      LDA   TIMSET,Y    ; read in the time to set
C086  9D 08 DC              STA   TODTN1,X    ; store to clock—hrs. first
C089  C8                    INY               ; for next TIMSET byte
C08A  CA                    DEX               ; for next clock byte (mins., secs., tenths)
C08B  10 F6                 BPL   SETLOP      ; set all four bytes of clock
C08D  60                    RTS
                                              ;
C08F  82 30 13  TIMSET      .BYTE $82,$30,$13,$0
                                              ; hrs., mins., secs., tenths for clock
                                              ; (02.30.13.0 p.m.)
                                              ; For a.m., subtract $80 from hrs. place.
C092  01        CLKFLG      .BYTE 1           ; clock display flag—display it (1) or don't
                                              ; display (0)
```

See also ALARM2, TOD1DL, TOD1RD, TOD2PR, TOD2ST.

Name

Produce a delay using an IRQ interrupt counter

Description

INTDEL uses the IRQ interrupt as an event timer.

Unless they're disabled, interrupt requests (IRQs) occur at regular intervals—once every 1/60 second to be exact—regardless of what's happening in the main program. This is the basis of this routine.

INTDEL updates a counter during each IRQ interrupt, thus freeing your main program to do other things. In other words, you no longer have to halt the current action to update a timer. Instead, you can wait until the ongoing activity is complete before checking the state of the timer.

For instance, if you're writing a joystick-controlled, timed, arcade-style game in which a player must defend his ground base from aerial invaders in the form of sprites. And these sprites, as is often the case, are interrupt-driven, meaning they're constantly moving regardless of what's happening in the rest of your program.

Now suppose the player needed to aim his artillery at an incoming attacker, but your program was off somewhere updating the timer. It could easily be curtains for the unfortunate player. But with this routine, you could allow the player to ward off the attacker before checking the timer.

Another practical application of an interrupt timer such as this one is in generating interrupt-driven music. Here, the interrupt timer typically determines the duration of a specific note.

Prototype

In **INTDEL**:

1. Disable IRQ interrupts with SEI.
2. Redirect the IRQ interrupt vector at 788 to DWEDGE.
3. Initialize the counter flag to a value of one, indicating the countdown is ongoing.
4. Set DELCTR to the delay time specified by DELAY. In the process, increment the high byte of DELCTR by one.
5. With the vector having been changed in step 2, reenable IRQ interrupts and RTS.

In DWEDGE:

1. Check CTRFLG to determine if a delay countdown is in progress.

2. If it isn't (CTRFLG = 0), exit the routine through the normal IRQ interrupt handler (in step 7).
3. Otherwise, decrement the low byte of the delay counter and exit through the normal IRQ interrupt handler, provided the low byte hasn't reached zero.
4. If the low byte has reached zero in step 3, then decrement the high byte of DELCTR as well.
5. If the resulting high byte has yet to reach zero, then exit through step 7.
6. Otherwise, store a value of zero to CTRFLG, indicating the countdown is complete.
7. Exit by executing the normal IRQ interrupts.

Explanation

The program below initially sets the two-byte interrupt timer (DELCTR) in **INTDEL** to 330 interrupts, or five and a half seconds, and the timer flag (CTRFLG) to 1. Then, within INLOOP, it prints a series of ten spade characters on the screen before checking the timer flag. If CTRFLG is 1, meaning the IRQ timer is still counting down, the program prints another ten spades.

When the timer finally reaches zero, CTRFLG itself becomes zero in $C041. This halts the main program, but not before the last ten spades have printed.

Note: As always, when redirecting the IRQ vector to your own routine, be sure you first disable the IRQ interrupts.

Routine

```
C000                ZP        =     251
C000                CHROUT    =     65490
C000                SPADE     =     97         ; ASCII value for spade character
C000                IRQVEC    =     788        ; vector to IRQ interrupt routine
C000                IRQNOR    =     59953      ; IRQNOR = 64101 on the 128—normal IRQ
                                               ; interrupt service routine
C000                DELAY     =     330        ; delay for 330 IRQ interrupts (5.5 secs.)
                                               ;
                                               ; Carry out an activity (INLOOP) until the
                                               ; interrupt delay finishes.
C000  20 13 C0      MAIN      JSR   INTDEL     ; setup the interrupt delay
C003  A9 61         MNLOOP    LDA   #SPADE     ; get the spade character
C005  A0 0A                   LDY   #10        ; initialize index for INLOOP
C007  20 D2 FF      INLOOP    JSR   CHROUT     ; print it
C00A  88                      DEY
C00B  D0 FA                   BNE   INLOOP     ; repeat INLOOP ten times
C00D  AD 47 C0                LDA   CTRFLG     ; is countdown complete?
C010  D0 F1                   BNE   MNLOOP     ; if not, then continue MNLOOP
C012  60                      RTS              ; we're finished
                                               ;
                                               ; Insert IRQ interrupt wedge for delay timer.
                                               ; Initialize flag and delay.
```

```
C013  78              INTDEL   SEI                 ; disable IRQ interrupts to change IRQ
                                                   ; vector
                                                   ; Then store the address of our routine into
                                                   ; IRQ vector.
C014  A9  30                   LDA    #<DWEDGE     ; low byte first
C016  8D  14  03               STA    IRQVEC
C019  A9  C0                   LDA    #>DWEDGE     ; then high byte
C01B  8D  15  [illegible]      STA    IRQVEC+1
C01E  A9  01                   LDA    #1           ; initialize CTRFLG to 1
C020  8D  47  C0               STA    CTRFLG
C023  A9  4A                   LDA    #<DELAY      ; initialize DELCTR, low byte first
C025  8D  48  C0               STA    DELCTR
C028  A2  01                   LDX    #>DELAY      ; then high byte
C02A  E8                       INX                 ; so high byte goes from one to zero on last
                                                   ; pass during countdown
C02B  [illegible]  49  C0      STX    DELCTR+1
C02E  58                       CLI                 ; We've reset the vector. Now reenable IRQ
                                                   ; interrupts and
C02F  60                       RTS                 ; exit setup.
                                                   ;
C030  AD  47  C0  DWEDGE       LDA    CTRFLG       ; check to see if countdown is ongoing
C033  F0  0F                   BEQ    EXIT         ; if not, exit through the normal IRQ
                                                   ; interrupt routines
C035  CE  [illegible]  C0      DEC    DELCTR       ; decrement low byte of delay counter
C038  D0  0A                   BNE    EXIT         ; if low byte hasn't turned over yet, exit
C03A  CE  49  C0               DEC    DELCTR+1     ; the low byte has reached zero, so decrease
                                                   ; counter high byte
C03D  D0  05                   BNE    EXIT         ; if high byte is not zero, exit
                                                   ; DELCTR has reached zero (both low and
                                                   ; high bytes).
C03F  A0  00                   LDY    #0
C041  8C  47  C0               STY    CTRFLG       ; to prevent further countdown
C044  4C  31  EA  EXIT         JMP    IRQNOR       ; service the standard IRQ routines
                                                   ;
C047  00          CTRFLG       .BYTE  0            ; flag is one while countdown continues, zero
                                                   ; when done
C048  00  00      DELCTR       .WORD  0            ; storage for two-byte interrupt delay counter
```

See also BYT1DL, BYT2DL, JIFDEL, KEYDEL, TOD1DL.

Name

Interrupt-driven music

Description

With **INTMUS**, you can enhance any programs—especially games—by adding background music that runs automatically.

Prototype

Before entering this routine, set up a table of note values which index frequencies from FREQTB (NOTES), a table containing the relative durations for each note in NOTES (NDURTB), and a table of the two-byte frequencies needed for the tune (FREQTB).

In the initialization routine (**INTMUS**):

1. Disable IRQ interrupts before changing the IRQ interrupt vector.
2. Redirect the IRQ interrupt vector to the music-playing routine (MAIN).
3. Set a note counter (NOTENM) to zero.
4. Clear the SID chip with **SIDCLR** and set the appropriate parameters for the chip (volume and attack/decay).
5. Initialize a duration counter (DURATE) for the first pass through MAIN.
6. Reenable IRQ interrupts and RTS.

Then, in MAIN:

1. Decrement the duration counter.
2. If it decrements to zero, get a note to play. Otherwise, allow the note that's currently playing to continue by exiting through the normal IRQ interrupt handler.
3. Assuming the duration counter reaches zero, get the note number and index the next note's duration using it.
4. Adjust the time each note plays by multiplying its duration by some factor (here, 8).
5. Store the result in the duration counter.
6. Get a note from the NOTES table and use it to index the corresponding two-byte frequency value in FREQTB. Store the frequency taken from FREQTB into the frequency registers for voice 1.
7. Ungate, and then gate, the waveform (here, a sawtooth waveform).

8. Increment the note counter and determine if all notes have played. If not, continue playing the tune. Otherwise, reinitialize the note counter to start the tune over.

Explanation

The principle behind interrupt-driven music is that you let the IRQ interrupt generated every 1/60 second determine when and how long each note is played.

After redirecting the IRQ vector to a music-playing routine (MAIN), the SID chip is set up and several counters are initialized. One of these counts how many notes have been played (NOTENM) while the other keeps up with how long the current note has played (DURATE).

Once IRQ interrupts are reenabled, MAIN is accessed during each IRQ interrupt. The first time this happens, a note based on a reference value (in NOTES) is selected from a table of frequencies (FREQTB) and stored in the frequency register for voice 1. At the same time, a duration time for the note is taken from another table (NDURTB) and stored in the duration counter (DURATE). Before exiting, the pointer to the next note (NOTENM) is incremented and the current note starts playing.

Each time the IRQ returns to MAIN thereafter, the duration counter decrements. When it reaches zero, the next note from NOTES gets stored into the frequency register, DURATE is reset for this note's duration, and the cycle repeats itself. When all notes have played, NOTENM becomes zero, and the tune starts over again.

In setting up the note (NOTES) and frequency (FREQTB) tables, the same method used in **MELODY** is used here. Each number in NOTES references a two-byte frequency value in FREQTB. Again, the frequencies listed in FREQTB are taken from the table of notes in the programmer's reference guide for either the 64 or 128. Expand FREQTB to include whatever notes your song calls for. If you like, you can even have **NOTETB** generate a complete frequency table for you.

After you've worked out the relative time spent playing each note with the values in NDURTB, you'll need to adjust the overall tempo of the song. The three ASLs at $C02F, for the current song, increase the tempo by a factor of eight. For each tune you play, you may need to add or take away one or more of these (ASLs) before the song sounds right.

Routine

```
C000                  IRQVEC     =      788           ; vector to IRQ interrupt routine
C000                  IRQNOR     =      59953         ; IRQNOR = 64101 on the 128
C000                  FRELO1     =      54272         ; starting address for the SID chip
C000                  FREHI1     =      54273         ; voice 1 high frequency
C000                  VCREG1     =      54276         ; voice 1 control register
C000                  ATDCY1     =      54277         ; voice 1 attack/decay register
C000                  SIGVOL     =      54296         ; SID chip volume register
                                                      ;
                                                      ; Set up an IRQ interrupt to play background
                                                      ; music.
C000  78              INTMUS     SEI                  ; disable IRQ interrupts to change the
                                                      ; vector
C001  A9 24                      LDA    #<MAIN        ; store the low byte of the IRQ wedge
C003  8D 14 03                   STA    IRQVEC
C006  A9 C0                      LDA    #>MAIN        ; and the high byte
C008  8D 15 03                   STA    IRQVEC+1
[redacted]  A9 00                LDA    #0
C00D  8D A1 C0                   STA    NOTENM        ; set pointer to first note in table
C010  20 A2 C0                   JSR    SIDCLR        ; clear the SID chip
C013  A9 0F                      LDA    #15           ; set the volume to maximum
C015  8D 18 D4                   STA    SIGVOL
C018  A9 1A                      LDA    #$1A          ; set attack/decay
C01A  8D 05 D4                   STA    ATDCY1
C01D  A9 01                      LDA    #1
C01F  8D A0 C0                   STA    DURATE        ; initialize duration counter for first pass
C022  58                         CLI                  ; with vector changed, reenable IRQ
                                                      ; interrupts
C023  60                         RTS
                                                      ;
                                                      ; Main actually plays the music.
C024  CE A0 C0        MAIN       DEC    DURATE        ; see if current note has finished playing
C027  D0 36                      BNE    EXIT          ; if not, allow it to finish
C029  AE A1 C0                   LDX    NOTENM        ; index to NOTES
C02C  BD 7B C0                   LDA    NDURTB,X      ; get the note's duration from a table
C02F  0A                         ASL                  ; multiply by 8 so each note lasts eight times
                                                      ; longer
C030  0A                         ASL
C031  0A                         ASL
C032  8D A0 C0                   STA    DURATE        ; and store it into the counter
C035  BD 62 C0                   LDA    NOTES,X       ; get index for FREQTB
C038  0A                         ASL                  ; double it since FRFQTB contains two-byte
                                                      ; addresses
C039  AA                         TAX                  ; to index FREQTB
C03A  BD 94 C0                   LDA    FREQTB,X      ; get low byte of note's frequency
C03D  8D 00 D4                   STA    FRELO1        ; store it in voice 1
C040  BD 95 C0                   LDA    FREQTB+1,X    ; get high byte of note's frequency
C043  8D 01 D4                   STA    FREHI1        ; store it in voice 1
C046  A9 20                      LDA    #%00100000    ; ungate sawtooth waveform
C048  8D 04 D4                   STA    VCREG1
C04B  A9 21                      LDA    #%00100001    ; gate waveform
C04D  8D 04 D4                   STA    VCREG1
C050  EE A1 C0                   INC    NOTENM        ; increase note counter
C053  AD A1 C0                   LDA    NOTENM
C056  C9 19                      CMP    #NMNOTE       ; determine if all notes have played
C058  90 05                      BCC    EXIT          ; if not, then continue
C05A  A9 00                      LDA    #0
C05C  8D A1 C0                   STA    NOTENM        ; if yes, start again with first note
C05F  4C 31 EA        EXIT       JMP    IRQNOR        ; exit through normal IRQ interrupt handler
                                                      ;
C062  02 02 04        NOTES      .BYTE  2,2,4,4,5,5,4,5,5,4,3,2
                                                      ; table of note indexes
C06E  03 02 02                   .BYTE  3,2,2,4,2,1,0,0,0,0,1,1,2
```

```
C07B                 NMNOTE   =       * - NOTES     ; number of notes
C07B  02  06  02   NDURTB   .BYTE  2,6,2,6,4,3,1,2,2,1,1,2,1,1,4,2
                                                     ; table of note durations
C08B  01  02  03            .BYTE  1,2,3,1,2,2,1,2,12
C094  C3  10  EF   FREQTB   .WORD 4291,5103,5728,6812,7647,8583
                                                     ; tabale of two-byte frequency values
C0A0  00           DURATE   .BYTE  0                 ; duration counter
C0A1  00           NOTENM   .BYTE  0                 ; note number counter
                                                     ;
                                                     ; Clear the SID chip.
C0A2  A9  00       SIDCLR   LDA    #0                ; fill with zeros
C0A4  A0  18                LDY    #24               ; as the offset from FRELO1
C0A6  99  00  D4   SIDLOP   STA    FRELO1,Y          ; store zero in each SID chip address
C0A9  88                    DEY                      ; for next lower address
C0AA  10  FA                BPL    SIDLOP            ; fill 25 bytes
C0AC  60                    RTS                      ; we're done
```

See also BEEPER, BELLRG, EXPLOD, MELODY, NOTETB, SIDCLR, SIDVOL, SIRENS.

IRQINT

Name

Set up an IRQ interrupt routine

Description

IRQINT redirects the IRQ interrupt vector to your own routine

Prototype

1. SEI to disable the IRQ interrupts.
2. Store the address of your custom IRQ routine into the IRQ interrupt vector.
3. Reenable the IRQ interrupts with a CLI and RTS.

Explanation

The program below demonstrates how this routine might be used. In it, **IRQINT** changes the IRQ vector to point to the routine WEDGE. This routine, in turn, checks the shift key flag, halting the current program if a shift key is being pressed. The shift keys include SHIFT, CTRL, and the Commodore key on the 64 and 128; and also CAPS LOCK and ALT on the 128.

Since WEDGE is accessed during each IRQ interrupt (every 1/60 second), you can halt almost anything run from BASIC—games, commands such as LIST, and so on.

Notice we rely on the Kernal routine SCNKEY rather than GETIN within our interrupt routine. Unlike GETIN, SCNKEY updates even while we're in the interrupt routine.

Note: It's important to disable IRQ interrupts, as we've done here, before changing the IRQ vector. If you skip this step and an IRQ interrupt occurs while the vector is being changed, your program could easily be sent to some meaningless address.

On the 128, your custom IRQ routine must be accessible from bank 15 since memory is configured for this bank prior to jumping through the IRQ vector.

Routine

```
C000             IRQVEC  =    788       ; vector to IRQ Interrupt vector
C000             IRQNOR  =    59953     ; IRQNOR = 64101 on the 128—normal IRQ
                                        ; interrupt handler
C000             SCNKEY  =    65439     ; Kernal routine to get a keypress
C000             SHFLAG  =    653       ; SHFLAG = 211 on the 128—shift key flag
                                        ;
                                        ; IRQ interrupt routine to pause on shift key.
C000  78         IRQINT  SEI            ; disable the IRQ interrupts before
                                        ; changing the vector
C001  A9 0D              LDA  #<WEDGE   ; point the IRQ vector to our routine, low
                                        ; byte first
```

```
C003  8D 14 03            STA   IRQVEC
C006  A9 C0               LDA   #>WEDGE      ; and then high byte
C008  8D 15 03            STA   IRQVEC+1
C00B  58                  CLI                ; reenable IRQ interrupts after changing
                                             ; the vector
C00C  60                  RTS
                                             ;
                                             ; Halt the program with SHIFT keypress.
C00D  AD 8D 02   WEDGE    LDA   SHFLAG       ; check the SHIFT flag
C010  F0 06               BEQ   FINIS        ; if SHIFT not pressed, then exit through
                                             ; normal IRQ routine
C012  20 9F FF            JSR   SCNKEY       ; update SHIFT flag
C015  4C 0D C0            JMP   WEDGE        ; and check if it's still pressed
C018  4C 31 EA   FINIS    JMP   IRQNOR       ; exit through the normal IRQ interrupt
                                             ; handler
```

See also NMIINT, RAS64, RAS128.

Name

Jiffy clock delay

Description

One- and two-byte delay routines, causing pauses of less than a millisecond to a few seconds, have been provided elsewhere in this book (**BYT1DL, BYT2DL**). There will be times, though, when you'll need a routine to produce an extended delay—on the order of several seconds to several minutes. **JIFDEL**, which relies on the jiffy clock to time this delay, is just such a routine.

Prototype

1. Enter this routine with the delay length (defined in jiffies as DELAYJ) in .A (low byte) and .X (high byte). The current jiffy clock reading (the low and middle bytes) are in zero page (in ZP).
2. Add the delay value to the jiffy clock reading in ZP.
3. Compare the resulting value to the current jiffy clock reading and return from the routine when they agree.

Explanation

JIFDEL is a straightforward and practical routine. First add the number of jiffies (1/60 second intervals) that you've specified in DELAYJ to the current jiffy clock reading and then wait until the clock reads this total.

As it's written, the routine only uses the lower two bytes of the three-byte clock. With these two bytes alone, a delay anywhere from 1/60 second (one jiffy) to 1092 seconds (65,535 jiffies or 18.2 minutes) can be carried out. If you need a program delay that extends for an even longer time than 18.2 minutes, add the high byte of the jiffy clock as well.

In the example program below, **JIFDEL** causes a delay of 600 jiffies—ten seconds—before incrementing the border color of the screen. Notice that most of the code for this program is setup required by **JIFDEL**. The lower two bytes of the current jiffy clock reading are stored into zero page. Before this can be done, IRQ interrupts must be disabled so the clock won't advance while it's being read. The last requirement is that the specified delay (DELAYJ) be passed to the routine in the accumulator (low byte) and the X register (high byte).

Routine

```
C000                CHROUT   =    65490
C000                ZP       =    251
C000                TIME     =    160         ; three-byte jiffy clock
C000                EXTCOL   =    53280       ; border color register
C000                DELAYJ   =    600         ; 600 jiffies (ten seconds)
                                              ;
                                              ; Cause the border color to change after a
                                              ; specified delay.
C000  78                     SEI              ; disable interrupts so clock doesn't advance
                                              ; while being read
C001  A5 A2                  LDA  TIME+2      ; store jiffy low byte in zero page
C003  85 FB                  STA  ZP
C005  A6 A1                  LDX  TIME+1      ; store middle byte also
C007  86 FC                  STX  ZP+1
C009  58                     CLI              ; we've got the current jiffy time, so reenable
                                              ; interrupts
C00A  A9 58                  LDA  #<DELAYJ    ; store low byte and high byte of jiffy delay
C00C  A2 02                  LDX  #>DELAYJ
C00E  20 15 C0               JSR  JIFDEL      ; carry out delay in .A and .X
C011  EE 20 D0               INC  EXTCOL      ; change the border color
C014  60                     RTS
                                              ;
                                              ; JIFDEL sets the jiffy clock with the delay in
                                              ; .A (low) and .X (middle).
C015  18            JIFDEL   CLC              ; add delay to current jiffy clock reading in
                                              ; zero page
C016  65 FB                  ADC  ZP          ; low byte first
C018  85 FB                  STA  ZP
C01A  8A                     TXA              ; now middle byte
C01B  65 FC                  ADC  ZP+1
                                              ; Determine whether DELAYJ has elapsed.
C01D  C5 A1         MIDBYT   CMP  TIME+1      ; check middle byte first
C01F  D0 FC                  BNE  MIDBYT      ; wait for middle byte to agree
C021  A5 FB                  LDA  ZP          ; now low byte
C023  C5 A2         LOWBYT   CMP  TIME+2
C025  D0 FC                  BNE  LOWBYT      ; wait for low byte to agree
C027  60                     RTS              ; previous time is equal to time plus delay
```

See also BYT1DL, BYT2DL, INTDEL, KEYDEL, TOD1DL, JIFFRD, JIFPRT, JIFSET.

Name

Read the jiffy clock

Description

JIFFRD does more than just read the three-byte jiffy clock. This routine is integrated into a program in which a pair of timers are updated based on the current jiffy clock reading.

Prototype

1. Disable IRQ interrupts to prevent the clock from advancing while it's being read.
2. In a loop, read three bytes from the jiffy clock, storing them to a memory buffer. (Here, we actually add them to the current timer value for player 1 or 2.)
3. Reenable IRQ interrupts to restart the jiffy clock.

Explanation

It's a relatively simple matter to read the three-byte jiffy clock at location 160. You first disable IRQ interrupts to stop the clock, read the three bytes into a memory buffer, and reenable IRQ interrupts to restart the clock.

This routine offers additional features. It is part of a simulation in which two 3-byte jiffy timers are maintained—one for each of two players. Let's say you've brought your computer to a hockey game and you want to keep track of time of possession. When one team has the puck, press the 0 key. When the other team gets it, press the 1 key. The jiffy clock is reset to zero at the beginning of each event.

When a change of possession occurs (when the other key is pressed), the current jiffy clock reading is added to the appropriate timer, and the program begins timing the other team's turn. This continues—teams alternating turns—until the space bar, which exits the program, is pressed.

At the start of the program, both timers are initialized to zero in INITLP. The clock then begins at START after 0 or 1 is pressed. Pressing one of these keys causes a branch to INITTM where the jiffy clock is reset. The value of the ASCII keypress is then used in SETUPZ to load the address of the current team's timer from TABTIM into zero page.

Once the current team's timer address is in zero page, we jump to MAINLP where the third key—the space bar—becomes an acceptable entry. The 0 and 1 keys, at this point, cause a switch to occur. The timer for the previous team is updated in **JIFFRD**.

Within **JIFFRD**, we momentarily stop the jiffy clock with an SEI, add the current reading to the last team's timer, reset the clock, and start it again with a CLI. From here, provided the space bar isn't pressed, we branch to SETUPZ—where the current team's timer address is stored in zero page—and again jump to MAINLP. Notice that the structure of the program allows a team to repeat without corrupting the timers.

Note: In adding the jiffy clock to the timer in $C02F, the zero-page address for the jiffy clock must be expressed as a two-byte address (as $00A0). That's because the opcode form ADC *zero-page address*,Y doesn't exist in 6502/8502 machine language.

Routine

```
C000                GETIN    =     65508
C000                ZP       =     251
C000                TIME     =     160          ; three-byte jiffy clock
                                                ;
                                                ; Add to each player's timer when player
                                                ; switch occurs. Quit on space bar
C000  A0 05                  LDY   #5           ; initialize players' timers to zero
C002  A9 00                  LDA   #0
C004  99 5A C0      INITLP   STA   PLAYR1,Y
C007  88                     DEY
C008  10 FA                  BPL   INITLP       ; do all six bytes
                                                ;
C00A  20 E4 FF      START    JSR   GETIN        ; set the jiffy clock to zero with the first valid
                                                ; keypress
C00D  C9 30                  CMP   #48          ; does player 1 start the jiffy clock first?
C00F  F0 27                  BEQ   INITTM       ; initialize jiffy clock and put PLAYR1 in ZP
C011  C9 31                  CMP   #49          ; or does player 2 start it first?
C013  F0 23                  BEQ   INITTM       ; initialize jiffy clock and put PLAYR2 in ZP
C015  D0 F3                  BNE   START        ; it's neither, so get another keypress
                                                ;
C017  20 E4 FF      MAINLP   JSR   GETIN        ; main GETIN loop
C01A  C9 30                  CMP   #48          ; is it player 1's turn?
C01C  F0 0A                  BEQ   JIFFRD       ; add in jiffy clock to PLAYR2
C01E  C9 31                  CMP   #49          ; is it player 2's turn?
C020  F0 06                  BEQ   JIFFRD       ; add in jiffy clock to PLAYR1
C022  C9 20                  CMP   #32          ; is it SPACE?
C024  F0 02                  BEQ   JIFFRD       ; add in the last player's time and quit
C026  D0 EF                  BNE   MAINLP       ; if not 0, 1, or space, wait for another
                                                ; keypress
                                                ;
                                                ; JIFFRD reads the jiffy clock, adds the
                                                ; current value to PLAYR1 or
                                                ; PLAYR2, depending on which one just
                                                ; finished, and restarts the clock
C028  78            JIFFRD   SEI                ; stop the clock
C029  48                     PHA                ; save the player number as ASCII 48 or 49
C02A  18                     CLC                ; for subsequent addition
C02B  A0 02                  LDY   #2           ; add all three bytes of the jiffy clock to
                                                ; timer for PLAYR1 or PLAYR2
C02D  B1 FB         RDLOOP   LDA   (ZP),Y       ; get player's previous timer value
C02F  79 A0 00               ADC   $00A0,Y      ; add current jiffy clock reading to it
C032  91 FB                  STA   (ZP),Y       ; and store it back to PLAYR1 or PLAYR2
C034  88                     DEY                ; for next higher byte in the jiffy clock
```

```
C035  10  F6                  BPL    RDLOOP      ; do all three bytes
C037  68                      PLA                ; to properly maintain the stack with an
                                                 ; even number of PHA/PLA instructions
C038  48          INITTIM     PHA                ; save the player's number as ASCII 48 or
                                                 ; 49
C039  A9  00                  LDA    #0          ; reset timer
C03B  85  A0                  STA    TIME        ; do all three bytes
C03D  85  A1                  STA    TIME+1
C03F  85  A2                  STA    TIME+2
C041  68                      PLA                ; restore player number as ASCII 48 or 49
C042  58                      CLI                ; restart clock (only matters when SEI at
                                                 ; beginning of JIFFRD executes)
                                                 ;
C043  C9  20                  CMP    #32         ; quit on space (we've added in the last time
                                                 ; to PLAYR1 or PLAYR2)
C045  D0  01                  BNE    SETUPZ      ; if not space, set up ZP for next player
C047  60                      RTS
                                                 ;
                                                 ; Point ZP to the next player's timer.
C048  29  01      SETUPZ      AND    #1          ; to convert the ASCII response of 48/49 to
                                                 ; 0/1
C04A  0A                      ASL                ; double the number since we're dealing with
                                                 ; two-byte addresses (.WORDS)
C04B  A8                      TAY                ; index by .Y
C04C  B9  60  C0              LDA    TABTIM,Y    ; load low-byte address of PLAYR1 or
                                                 ; PLAYR2
C04F  85  FB                  STA    ZP          ; store in zero page
C051  C8                      INY                ; for next byte
C052  B9  60  C0              LDA    TABTIM,Y    ; load high-byte address of PLAYR1 or
                                                 ; PLAYR2
C055  85  FC                  STA    ZP+1        ; and store also
C057  4C  17  C0              JMP    MAINLP      ; and wait for another key
                                                 ;
C05A  00  00  00  PLAYR1      .BYTE  0,0,0       ; three-byte timer for player 1
C05D  00  00  00  PLAYR2      .BYTE  0,0,0       ; three-byte timer for player 2
C060  5A  C0  5D  TABTIM      .WORD PLAYR1,PLAYR2
                                                 ; address pointers to each player's timer
```

See also JIFDEL, JIFPRT, JIFSET.

Name

Print the jiffy clock reading

Description

This routine allows you to use the three-byte jiffy clock as a timepiece. **JIFPRT** displays the current jiffy clock reading on the screen in an hours/minutes/seconds/jiffies format.

Prototype

1. Initialize a place counter (CLKCTR) to zero for the ASCII clock frame (CLOCK).
2. Store the address of this clock frame into zero page (as ZT).
3. Disable IRQ interrupts to prevent the jiffy clock from advancing while it's being read.
4. Read the current three-byte jiffy clock reading and store it in zero page (ZP). Reenable IRQ interrupts.
5. Load .X with an index to the subtrahends table (TB3SUB) so that it initially points to the low byte of the largest subtrahend (the low byte $80 of 2160000/$20F580).
6. Perform a conversion of the jiffy clock reading to an hours/minutes/seconds/hundredths-of-seconds format by repeated subtraction. Store the ASCII equivalent of each digit into the clock frame.
7. After each digit has been converted to ASCII, a check of CLKCTR tells us whether the next digit's place in the clock frame is even or odd. On even-digit places, the zero-page pointer to the clock frame is incremented by one, which places us beyond the colons or the decimal in the frame.
8. When the ASCII clock has been completed, print it and return from the routine.

Explanation

In the following program, a formatted jiffy clock is continually printed at the home position with **JIFPRT** until a key is pressed.

The three-byte jiffy clock at 160–162 is a 24-hour cascade timer, updated by the operating system. Unlike most other pointers and values in memory, the high byte of the jiffy clock (160) is actually lowest in memory.

The jiffy clock increments every 1/60 second, a unit of time called a *jiffy*. The low byte at location 162 counts 256 jiffies (4.27 seconds) before the middle byte, location 161, increments. When the middle byte reaches 256 (after 18.2 min-

utes), the high byte at location 160 counts forward by one.

In **JIFPRT**, after storing the current jiffy clock reading in zero page (ZP), it's converted to an hours/minutes/seconds/hundredths-of-seconds format and stored as ASCII into CLOCK. This conversion is done by using a subtraction method much like the two-byte conversion routine discussed in **CNUMOT**, only in this case it's done for a three-byte number. In very general terms, the current three-byte jiffy clock reading is divided by the eight three-byte numbers in TB3SUB. Each division, following conversion to ASCII at $C05D, yields another digit within CLOCK. We begin with the highest, or tens-of-hours place, and work down to the lowest, or sixtieths-of-seconds place.

Notice that, before running the program, CLOCK already contains the colons and the decimal used in the screen display. This setup is referred to as a *clock frame*. By prepositioning the colons and decimal point, we avoid having to write code to print them ourselves within **JIFPRT**. At the same time, however, we have to insure that we don't overwrite them when we store the ASCII digits to CLOCK. And this is where the CLOCK position counter, or CLKCTR, comes into play.

After storing each ASCII digit to CLOCK, we check to see whether the next position in clock, as maintained in CLKCTR, is even or odd (see $C05F–$C071). If CLKCTR tells us that the next position is even (the carry flag is clear after the LSR in $C069), we increment by one the zero-page pointer to the clock frame (in ZT) so that we skip over the colon or decimal which follows.

Once the clock frame has been constructed, it's a simple matter to print its ASCII contents in PRTCLK.

Routine

```
C000                CHROUT  =    65490
C000                GETIN   =    65508
C000                ZP      =    251
C000                ZT      =    155         ; two zero-page locations, normally used in
                                             ; tape loads
C000                TIME    =    160         ; three-byte jiffy clock
                                             ;
                                             ; Print the current jiffy clock reading. Hit any
                                             ; key to stop.
C000  A9 93         CLRCHR  LDA  #147        ; clear the screen
C002  20 D2 FF              JSR  CHROUT
C005  A9 13         JIFLOP  LDA  #19         ; HOME the cursor
C007  20 D2 FF              JSR  CHROUT
C00A  20 13 C0              JSR  JIFPRT      ; read and print the jiffy clock
C00D  20 E4 FF              JSR  GETIN       ; get a keypress
```

```
C010  F0 F3              BEQ   JIFLOP        ; if no keypress, do it all again
C012  60                 RTS
                                             ;
                                             ; JIFPRT reads and prints the jiffy clock.
C013  A9 00     JIFPRT   LDA   #0            ; initialize a place counter within our
                                             ; ASCII clock frame
C015  8D A9 C0           STA   CLKCTR
                                             ; Store the high and low bytes of our
                                             ; ASCII clock frame to zero page.
C018  A9 9D              LDA   #<CLOCK       ; low byte first
C01A  85 9B              STA   ZT
C01C  A2 C0              LDX   #>CLOCK       ; then high byte
C01E  86 9C              STX   ZT+1
                                             ;
C020  78        JIFFRD   SEI                 ; prevent the jiffy clock from advancing
                                             ; while it's being read
C021  A0 02              LDY   #2            ; as a index for LOOP
C023  B9 A0 00  LOOP     LDA   TIME,Y        ; store current jiffy clock reading in zero
                                             ; page
C026  99 FB 00           STA   ZP,Y
C029  88                 DEY
C02A  10 F7              BPL   LOOP
C02C  58                 CLI                 ; we've got the reading, so reenable IRQ
                                             ; interrupts
                                             ;
                                             ; Now convert clock reading in ZP to ASCII
                                             ; and store it in the ASCII clock.
C02D  A2 15              LDX   #21           ; index to TB3SUB table; initially points to
                                             ; low byte of 2160000
C02F  A0 FF     INITCT   LDY   #255          ; initialize counter for each digit's place
C031  C8        SUBTLP   INY                 ; begin subtraction loop, counter starts with
                                             ; zero
C032  A5 FD              LDA   ZP+2
C034  48                 PHA                 ; save the low byte of the current jiffy
                                             ; clock reading
C035  38                 SEC
C036  FD 85 C0           SBC   TB3SUB,X      ; subtract low byte of subtrahend from low
                                             ; byte of clock value
C039  85 FD              STA   ZP+2          ; store result in zero page
C03B  A5 FC              LDA   ZP+1          ; do the same with middle byte
C03D  48                 PHA                 ; save the middle byte of the current jiffy
                                             ; clock reading
C03E  FD 86 C0           SBC   TB3SUB+1,X    ; subtract subtrahend's middle byte from
                                             ; clock's middle byte
C041  85 FC              STA   ZP+1          ; and store the result
C043  A5 FB              LDA   ZP            ; and once again with the high byte
C045  48                 PHA                 ; save the high byte of the current jiffy
                                             ; clock reading
C046  FD 87 C0           SBC   TB3SUB+2,X    ; subtract high byte of subtrahend from
                                             ; clock's high byte
C049  85 FB              STA   ZP            ; and store the result
C04B  90 06              BCC   DONE          ; subtraction gave number less than 0 so
                                             ; we're done
C04D  68                 PLA                 ; restore the stack
C04E  68                 PLA
C04F  68                 PLA
C050  4C 31 C0           JMP   SUBTLP        ; and continue subtraction
                                             ; Restore high, middle, and low bytes to
                                             ; values before we dropped below zero.
C053  68        DONE     PLA                 ; pull high byte of clock reading
C054  85 FB              STA   ZP            ; and store it
C056  68                 PLA                 ; pull middle byte of clock reading
C057  85 FC              STA   ZP+1          ; and store it also
C059  68                 PLA                 ; pull low byte of clock reading
```

```
C05A  85  FD               STA   ZP+2       ; and store it also
C05C  98                   TYA              ; put digit's place counter into .A
                                            ; Convert digit's place counter to ASCII.
C05D  09  30               ORA   #48        ; effectively add 48 to get an ASCII digit
C05F  AC  A9  C0           LDY   CLKCTR     ; get the current clock place counter
C062  EE  A9  C0           INC   CLKCTR     ; update it for the next place
C065  91  9B               STA   (ZT),Y     ; store current ASCII digit into the clock
                                            ; frame
C067  C8                   INY              ; determine whether the next place is even
                                            ; or odd
C068  98                   TYA              ; shift the number right and check the
                                            ; carry flag
C069  4A                   LSR
C06A  B0  06               BCS   DECRIT     ; branch occurs with odd numbers
                                            ; If even, increment the clock frame pointer
                                            ; beyond the colon or decimal.
C06C  E6  9B               INC   ZT         ; increment low byte pointer
C06E  D0  02               BNE   DECRIT
C070  E6  9C               INC   ZT+1       ; and the high byte if the low byte wraps
C072  CA          DECRIT   DEX              ; decrement .X three times since three-byte
                                            ; entries in subtrahend table
C073  CA                   DEX
C074  CA                   DEX
C075  10  B8               BPL   INITCT     ; handle the next digit's place
                                            ;
                                            ; Now print the clock frame.
C077  A0  00               LDY   #0         ; as an index for PRTCLK
C079  B9  9D  C0  PRTCLK   LDA   CLOCK,Y    ; get each character from clock
C07C  F0  06               BEQ   EXIT       ; if zero byte, we're done
C07E  20  D2  FF           JSR   CHROUT     ; print each character from clock
C081  C8                   INY              ; next character
C082  D0  F5               BNE   PRTCLK     ; branch always
C084  60          EXIT     RTS
                                            ;
                                            ; A table of three-byte subtrahends follows.
C085  01  00  00  TB3SUB   .BYTE $1,$0,$0,$A,$0,$0,$3C,$0,$0,$58,$2,$0
C091  10  0E  00           .BYTE $10,$E,$0,$A0,$8C,$0,$C0,$4B,$3,$80,$F5,$20
C09D  20  20  3A  CLOCK    .ASC  "  : : ."  ; clock frame
C0A8  00                   .BYTE 0          ; terminator byte
C0A9  00          CLKCTR   .BYTE 0          ; position counter within the clock frame
```

See also JIFDEL, JIFFRD, JIFSET.

Name

Set the jiffy clock

Description

Since time is never expressed in binary format in everyday usage, the jiffy clock—a three-byte, 24-hour cascade timer—is awkward for those of us who are accustomed to an hours/minutes/seconds decimal format. **JIFSET** allows you to set this clock to a particular time that is defined in this more conventional, decimal form.

Prototype

1. Before entering this routine, define the time for the jiffy clock in an hours/minutes/seconds/hundredths-of-seconds format (in TIMSET).
2. Initialize a digit's place counter (CLRCTR) to 7 for a 7–0 count (the jiffy clock reads to eight digits).
3. Disable IRQ interrupts to prevent the jiffy clock from advancing while it's being set.
4. Clear the jiffy clock by storing a zero to its three bytes.
5. Initialize the X register to zero so that it initially points to the low byte of the smallest addend (the low byte of \$000001) in a table of addends (TB3ADD).
6. In RDSET, perform a three-byte conversion of the intended time (TIMSET) to the format used by the jiffy clock, set the clock, then reenable interrupts and return to the calling program.

Explanation

JIFSET sets the jiffy clock time to the value in TIMSET. In the example, time is set to 18:02:45.00. (The equivalent BASIC statement would be TI\$ = "180245".)

The approach taken in converting TIMSET to a jiffy-clock format is the opposite of that used in **JIFPRT**, which converts the clock reading to an hours/minutes/seconds/hundredths-of-seconds format.

Instead of using a subtraction method to do this conversion, we use addition here. Roughly speaking, each digit within TIMSET—beginning with the most significant digit, or the tenths-of-hours' place—is multiplied by the corresponding three-byte number in TB3ADD. This process continues until all digits have been accounted for. Accomplish each so-called multiplication by first storing the current digit in a counter

(CLKCTR) and then repeatedly adding the respective three-byte addend until the counter decrements to zero.

The interim result of each three-byte addition can be stored into the three memory locations used by the jiffy clock. This is possible since we have earlier disabled the IRQ interrupts which would ordinarily update the jiffy clock.

Routine

```
C000                ZP       =     251
C000                TIME     =     160           ; three-byte jiffy clock
                                                 ;
                                                 ; Set the jiffy clock to TIMSET
C000  A9 07         JIFSET   LDA   [redacted]    ; initialize a place counter
C002  8D 5C  C0              STA   CLKCTR
C005  78            JIFFRD   SEI                 ; prevent the jiffy clock from advancing
                                                 ; while it's being set
C006  A2 00                  LDX   #0            ; clear jiffy clock to zero and initialize .X
                                                 ; for ADDLOP
C008  86 A0                  STX   TIME
C00A  86 A1                  STX   TIME+1
C00C  86 A2                  STX   TIME+2
C00E  A0 00                  LDY   #0            ; as an index in TIMSET
C010  B9 54  C0     RDSET    LDA   TIMSET,Y      ; get a byte from TIMSET
C013  F0 1A                  BEQ   NEXTPL        ; if zero, skip ADDLOP
C015  A8                     TAY                 ; use .Y as an addition counter
C016  18            ADDLOP   CLC                 ; for addition
C017  A5 A2                  LDA   TIME+2        ; get the clock low byte
C019  7D 3C  C0              ADC   TB3ADD,X      ; add low byte of three-byte table entry
C01C  85 A2                  STA   TIME+2        ; store it in the clock
C01E  A5 A1                  LDA   TIME+1        ; do the same for clock middle byte
C020  7D 3D  C0              ADC   TB3ADD+1,X
C023  85 A1                  STA   TIME+1
C025  A5 A0                  LDA   TIME          ; do the same for clock high byte
C027  7D 3E  C0              ADC   TB3ADD+2,X
C02A  85 A0                  STA   TIME
C02C  88                     DEY                 ; decrement addition counter
C02D  D0 E7                  BNE   ADDLOP        ; repeat ADDLOP until respective TIMSET
                                                 ; digit is zero
C02F  E8            NEXTPL   INX                 ; for next three-byte entry in TB3ADD
C030  E8                     INX
C031  E8                     INX
C032  CE 5C  C0              DEC   CLKCTR        ; for next digit in TIMSET
C035  AC 5C  C0              LDY   CLKCTR
C038  10 D6                  BPL   RDSET         ; have all digits been handled?
C03A  58            EXIT     CLI                 ; we've set the jiffy clock, so reenable IRQ
                                                 ; interrupts
C03B  60                     RTS                 ; we're done
                                                 ;
                                                 ; three-byte table of addends
C03C  01 00  00     TB3ADD   .BYTE $1,$0,$0,$A,$0,$0,$3C,$0,$0,$58,$2,$0
C048  10 0E  00              .BYTE $10,$E,$0,$A0,$8C,$0,$C0,$4B,$3,$80,$F5,$20
C054  01 08  00     TIMSET   .BYTE 1,8,0,2,4,5,0,0
                                                 ; jiffy clock setting
C05C  00            CLKCTR   .BYTE 0             ; position counter within TIMSET
```

See also JIFDEL, JIFFRD, JIFPRT.

Name

Read both joysticks separately

Description

This routine reads both joysticks and returns a total of four values: the position of each stick (up, down, left, or right) and the state of the fire button for each joystick. The example routine contains a complete two-player game.

Prototype

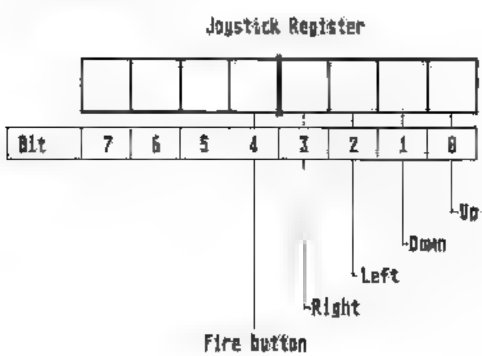
1. Load .Y with 1, as an index.
2. Load .A indexed by .Y from CIAPRA, the joystick register.
3. Exclusive-OR with %00010000 and then AND with %00010000, to isolate the bit that echoes the fire button.
4. Store this value in FIRE2, indexed by .Y.
5. LDA CIAPRA,Y again.
6. This time, EOR with %00001111 and then AND with %00001111.
7. Store the result in JOY2,Y.
8. Decrement .Y and branch back to step 2 while it's positive.

Explanation

There are two registers on the 64 and 128 that tell you the status of the joystick ports, locations 56320 and 56321 ($DC00–$DC01). These registers are called CIAPRA and CIAPRB—CIA data port A and port B. Unfortunately, the values you find here are doubly backwards.

The first way they're backwards is the labeling of the joystick port and the registers. Register B ($DC01) is joystick port 1. Register A ($DC00) is port 2. To read the first joystick, check the second register and vice versa.

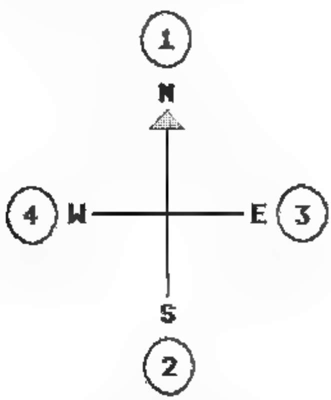
The second way they're backwards is the way the bits operate:



You might think that if the joystick is pushed to the left, bit 2 would be on and you'd see a value of $04 in the register. What really happens is that a %1 means the switch is off and %0 means it's on. So %xxx11110 means the joystick is being pushed forward.

The **JOY2SE** subroutine allows for the first problem by putting the JOY2 byte before JOY1, and FIRE2 before FIRE1 in memory (see locations $C0F7–$C0FA below). It solves the second problem by EORing the value with 15 or 16, then ANDing with 15 or 16. The result is a 16 in FIRE2 or FIRE1 if the fire button is down and a 0 if it's not. The value in JOY2 or JOY1 is 1, 2, 4, 8, or some combination of the numbers for diagonals (up and right would be 1 plus 8, for example).

The example program is a classic computer game. There are two players, each of whom has a joystick for moving. If a player doesn't touch the joystick, that player's character continues moving in the same direction. If the joystick is moved, the character changes direction (north, south, east, or west):



Each player leaves behind a trail, which marks the spaces the character (the worm) has previously traveled over. You can move into new territory, but if you hit a trail (or the edge of the screen), your worm dies, and points are awarded to your opponent.

The game as it appears is complete. But it could be improved. For example, after a crash, you could add the **EXPLOD** routine for a sound effect. The hearts and exclamation points that make up the worms could be improved with custom characters (see **CHRDEF** for an example of redefined characters).

Note to 128 users: Pressing the fire button on the 128 makes the computer act as if the F8 key was pressed. Thus, you may find that when the game ends, you're in the ML monitor. To prevent this, enter the line KEY8,"" before you play the game (normally, F8 is predefined to print MONITOR).

Routine

```
C000                ZP       =    $FB
C000                JIF      =    $A2          ; low byte of jiffy clock
C000                NDX      =    198          ; index to keyboard buffer (use 208 on
                                               ; the 128)
C000                CHCOLR   =    646          ; use 241 on the 128
C000                BGCOLR   =    53281
C000                CHROUT   =    $FFD2
C000                LINPRT   =    $BDCD        ; use $8E32 on the 128
C000                WM1      =    1040
C000                WM2      =    2000
C000                CIAPRA   =    56320
                                               ;
C000  20 FB C0               JSR  PREP         ; initial one-time setup for variables
C003  20 16 C1      ROUND    JSR  START        ; setup for the beginning of a round
C006  20 B3 C1               JSR  PUTIT        ; POKE a character to the screen
C009  20 DD C0      FLAG     JSR  JOY2SE       ; read the joysticks
C00C  AD FA C0               LDA  FIRE1        ; wait for the fire button
C00F  0D F9 C0               ORA  FIRE2        ; either one can start the game
C012  F0 F5                  BEQ  FLAG         ; keep looping until fire
C014  20 60 C0      MAINLP   JSR  SETDIR       ; set the direction
C017  20 B3 C1               JSR  PUTIT        ; put the character on the screen
C01A  A5 A2                  LDA  JIF
C01C  69 0A                  ADC  #10          ; delay is jiffy clock + 10
C01E  C5 A2         DLAY     CMP  JIF          ; compare it
C020  D0 FC                  BNE  DLAY         ; go back
C022  EE B1 C1               INC  POINTS       ; add one to the current round's points
C025  D0 03                  BNE  LY
C027  EE B2 C1               INC  POINTS+1     ; INC the high byte, if necessary
C02A  C0 00         LY       CPY  #0           ; does .Y hold a zero?
C02C  F0 E6                  BEQ  MAINLP       ; yes, keep going because neither player hit a
                                               ; wall
                                               ;
                                               ; end of a round
C02E  C0 02                  CPY  #2           ; did both players crash?
C030  F0 D1                  BEQ  ROUND        ; yes—no points, no penalty
C032  AD D9 C1               LDA  LOSER        ; either 0 or 2 for the loser
C035  49 02                  EOR  #2           ; flip 0 and 2, now it's the winner
C037  A8                     TAY               ; .Y holds the winner
C038  18                     CLC               ; get ready to add points
C039  AD B1 C1               LDA  POINTS       ; low byte of points
C03C  79 0D C1               ADC  P1SCOR,Y     ; add to the score
C03F  99 0D C1               STA  P1SCOR,Y     ; and store it
C042  AD B2 C1               LDA  POINTS+1     ; high byte
C045  79 0E C1               ADC  P1SCOR+1,Y   ; add it
C048  99 0E C1               STA  P1SCOR+1,Y   ; store it
                                               ;
C04B  AD D9 C1               LDA  LOSER        ; 0 or 2 again
C04E  4A                     LSR               ; make it 0 or 1
C04F  AA                     TAX
C050  DE 11 C1               DEC  P1WORM,X     ; one less worm for the loser (P1 or P2)
C053  F0 03                  BEQ  QUIT         ; if it's zero, quit
C055  4C 03 C0               JMP  ROUND        ; else, do another round
C058  20 16 C1      QUIT     JSR  START        ; print the final score
```

```
C05B  A9 00            LDA   #0          ; clear out
C05D  85 C6            STA   NDX         ; the keyboard buffer
C05F  60               RTS               ; and quit
                                         ;
                                         ; SETDIR does two things—continue the
                                         ; current path and set a new one.
C060  A0 01     SETDIR LDY   #1          ; index to P1DIR/P2DIR
C062  A2 02            LDX   #2          ; index to P1POS/P2POS
C064  B9 AF C1  CHKD   LDA   P1DIR,Y     ; get the number (1-4, for north, south, east,
                                         ; west)
C067  C9 01     CHK1   CMP   #1          ; north
C069  D0 10            BNE   CHK2        ; no, check south
C06B  BD AB C1         LDA   P1POS,X     ; yes, it is north
C06E  38               SEC               ; so move up (-40 in screen memory)
C06F  E9 28            SBC   #40         ; subtract
C071  9D AB C1         STA   P1POS,X     ; store
C074  B0 34            BCS   TRYNEX
C076  DE AC C1         DEC   P1POS+1,X   ; if carry clear, DEC the high byte
C079  10 2F            BPL   TRYNEX
C07B  C9 02     CHK2   CMP   #2          ; check for south
C07D  D0 10            BNE   CHK3        ; not south
C07F  BD AB C1         LDA   P1POS,X
C082  18               CLC
C083  69 28            ADC   #40         ; add 40
C085  9D AB C1         STA   P1POS,X
C088  90 20            BCC   TRYNEX
C08A  FE AC C1         INC   P1POS+1,X
C08D  10 1B            BPL   TRYNEX      ; branch always
                                         ;
C08F  C9 03     CHK3   CMP   #3          ; east, perhaps
C091  D0 0A            BNE   WEST        ; definitely west
C093  FE AB C1         INC   P1POS,X     ; add one to head east
C096  D0 12            BNE   TRYNEX
C098  FE AC C1         INC   P1POS+1,X
C09B  10 0D            BPL   TRYNEX
C09D  BD AB C1  WEST   LDA   P1POS,X
C0A0  E9 01            SBC   #1          ; carry is always set if we get this far
C0A2  9D AB C1         STA   P1POS,X
C0A5  B0 03            BCS   TRYNEX
C0A7  DE AC C1         DEC   P1POS+1,X
                                         ;
C0AA  CA        TRYNEX DEX               ; .X counts down two
C0AB  CA               DEX
C0AC  88               DEY
C0AD  10 B5            BPL   CHKD
                                         ;
C0AF  20 DD C0         JSR   JOY2SE      ; check the joystick
C0B2  AE F8 C0         LDX   JOY1        ; this will be a number 0–15
C0B5  F0 08            BEQ   SKIPIT
C0B7  BD CD C0         LDA   NSEW,X      ; find north, south, east, west
C0BA  F0 03            BEQ   SKIPIT
C0BC  8D AF C1         STA   P1DIR       ; direction for P1
C0BF  AE F7 C0  SKIPIT LDX   JOY2        ; look at player 2
C0C2  F0 08            BEQ   SKIP2
C0C4  BD CD C0         LDA   NSEW,X      ; find north, south, east, west again
C0C7  F0 03            BEQ   SKIP2
C0C9  8D B0 C1         STA   P2DIR       ; direction for P2
C0CC  60        SKIP2  RTS
C0CD  00 01 02  NSEW   .BYTE 0,1,2,0,4,0,0,0,3
C0D6  00 00 00         .BYTE 0,0,0,0,0,0
                                         ;
C0DD  A0 01     JOY2SE LDY   #1          ; index for checking 0 and 1
C0DF  B9 00 DC  JOYLP  LDA   CIAPRA,Y    ; joystick A (number 2) or B (number 1)
```

```
C0E2  49  10                 EOR   #16          ; flip bit 4
C0E4  29  10                 AND   #16          ; and isolate it
C0E6  99  F9  C0             STA   FIRE2,Y      ; store in the table
C0E9  B9  00  DC             LDA   CIAPRA,Y     ; check the joystick again
C0EC  49  0F                 EOR   #15          ; flip bits 0-3
C0EE  29  0F                 AND   #15          ; and mask off the high nybble
C0F0  99  F7  C0             STA   JOY2,Y       ; store the result
C0F3  88                     DEY                ; count down
C0F4  10  E9                 BPL   JOYLP        ; until .Y is  1
C0F6  60                     RTS
                                                ;
C0F7  00           JOY2      .BYTE 0
C0F8  00           JOY1      .BYTE 0
C0F9  00           FIRE2     .BYTE 0
C0FA  00           FIRE1     .BYTE 0
                                                ;
C0FB  A2  05       PREP      LDX   #PSIZ        ; copy the table PTAB
C0FD  BD  07  C1   PLOOP     LDA   PTAB,X       ; get the number
C100  9D  0D  C1             STA   P1SCOR,X     ; store it
C103  CA                     DEX                ; count down
C104  10  F7                 BPL   PLOOP        ; to -1 before
C106  60                     RTS                ; returning
C107  00  00  00   PTAB      .BYTE 0,0,0,0,5,5
                                                ; two 2-byte scores, plus five worms each
C10D               PSIZ      =     * PTAB-1     ; the size of the table
                                                ; which is copied to the variables below
C10D  00  00       P1SCOR    .BYTE 0,0          ; score
C10F  00  00       P2SCOR    .BYTE 0,0
C111  05           P1WORM    .BYTE 5            ; number of worms left
C112  05           P2WORM    .BYTE 5
C113  53           P1CH      .BYTE 83           ; screen code for heart
C114  00                     .BYTE 0            ; this byte is deliberately left blank
C115  21           P2CH      .BYTE 33           ; screen code for exclamation point
                                                ;
C116  A2  07       START     LDX   #RSIZ        ; copy the table RTAB
C118  BD  A3  C1   RLOOP     LDA   RTAB,X       ; get a number
C11B  9D  AB  C1             STA   P1POS,X      ; copy it
C11E  CA                     DEX                ; count down
C11F  10  F7                 BPL   RLOOP        ; until .X is -1
                                                ;
C121  A9  01                 LDA   #1           ; color code for white
C123  8D  86  02             STA   CHCOLR       ; character color
C126  8D  21  D0             STA   BGCOLR       ; background color
C129  A9  93                 LDA   #$93         ; clear screen character
C12B  20  D2  FF             JSR   CHROUT       ; print it
C12E  A9  0C                 LDA   #12          ; medium gray
C130  8D  21  D0             STA   BGCOLR       ; background color (do this to allow for
                                                ; version 2 64s)
C133  A9  04                 LDA   #4           ; purple
C135  8D  86  02             STA   CHCOLR
C138  A9  0D                 LDA   #13          ; <RETURN>
C13A  20  D2  FF             JSR   CHROUT
C13D  A9  DB                 LDA   #219         ; picket fence character
C13F  A2  29                 LDX   #41
C141  20  9C  C1             JSR   PRLP         ; print it .X number of times
C144  A2  15                 LDX   #21          ; repeat the next loop 21 times
C146  A9  9D       EDGES     LDA   #157         ; cursor left
C148  20  D2  FF             JSR   CHROUT       ; backup twice
C14B  20  D2  FF             JSR   CHROUT
C14E  A9  11                 LDA   #17          ; cursor down
C150  20  D2  FF             JSR   CHROUT
C153  A9  DB                 LDA   #219         ; picket fence again
C155  20  D2  FF             JSR   CHROUT
C158  20  D2  FF             JSR   CHROUT
```

```
C15B  CA                     DEX
C15C  D0  E8                 BNE   EDGES      ; print the edges
C15E  A2  27                 LDX   #39        ; now finish the bottom row
C160  20  9C  C1             JSR   PRLP       ; .A still holds the T shape
                                              ;
C163  A9  06                 LDA   #6         ; blue
C165  8D  86  02             STA   CHCOLR     ; character color blue
C168  AE  0F  C1             LDX   P2SCOR     ; get ready to print
C16B  AD  10  C1             LDA   P2SCOR+1   ; the score of player 2
C16E  20  CD  BD             JSR   LINPRT     ; print it
C171  A9  13                 LDA   #19        ; home (to print player 1's score)
C173  20  D2  FF             JSR   CHROUT
C176  AE  0D  C1             LDX   P1SCOR     ; low byte
C179  AD  0E  C1             LDA   P1SCOR+1   ; high byte
C17C  20  CD  BD             JSR   LINPRT
                                              ;
                                              ; Finish up by poking the number of
                                              ; remaining worms to the screen.
C17F  AD  13  C1             LDA   P1CH       ; the character
C182  AE  11  C1             LDX   P1WORM     ; number of worms left
C185  F0  06                 BEQ   OOPS1
C187  9D  10  04   POK1      STA   WM1,X
C18A  CA                     DEX
C18B  D0  FA                 BNE   POK1
C18D  AD  15  C1   OOPS1     LDA   P2CH       ; the character for P2
C190  AE  12  C1             LDX   P2WORM     ; how many worms are left?
C193  F0  06                 BEQ   OOPS2
C195  9D  D0  07   POK2      STA   WM2,X
C198  CA                     DEX              ; count down
C199  D0  FA                 BNE   POK2
C19B  60           OOPS2     RTS              ; end
                                              ;
C19C  20  D2  FF   PRLP      JSR   CHROUT     ; this routine prints the character in .A
C19F  CA                     DEX              ; counts down
C1A0  D0  FA                 BNE   PRLP       ; and repeats .X times
C1A2  60                     RTS
C1A3  9A  05  D6   RTAB      .WORD 1434,1494  ; starting positions
C1A7  03  04                 .BYTE 3,4        ; directions
C1A9  00  00                 .BYTE 0,0        ; initial points
C1AB               RSIZ      =     *-RTAB-1
C1AB  00  00       P1POS     .WORD 0          ; position of player 1
C1AD  00  00       P2POS     .WORD 0          ; and player 2
C1AF  00           P1DIR     .BYTE 0          ; direction of P1
C1B0  00           P2DIR     .BYTE 0          ; and P2
C1B1  00  00       POINTS    .BYTE 0,0        ; points for a round
                                              ;
C1B3  A2  03       PUTIT     LDX   #3         ; first get the addresses of the characters
C1B5  BD  AB  C1   KELP      LDA   P1POS,X    ; put the positions into ZP
C1B8  95  FB                 STA   ZP,X       ; two pointers
C1BA  CA                     DEX              ; and loop
C1BB  10  F8                 BPL   KELP       ; down to zero
                                              ;
C1BD  A0  00                 LDY   #0         ; .Y is going to indicate a winner if a collision
                                              ; occurs
C1BF  A2  02                 LDX   #2         ; offset for the characters
C1C1  A1  FB       LOOK      LDA   (ZP,X)     ; check the current location
C1C3  C9  20                 CMP   #32        ; if it's a space
C1C5  F0  08                 BEQ   WHEW       ; we're safe
C1C7  C8                     INY              ; else, there's a problem
C1C8  A9  56                 LDA   #86        ; X like character
C1CA  8E  D9  C1             STX   LOSER      ; .X holds the loser
C1CD  D0  03                 BNE   STORIT     ; branch always
```

```
C1CF  BD  13  C1  WHEW     LDA   PICH,X     ; get one of the characters
C1D2  81  FB      STORIT   STA   (ZP,X)     ; and store it to the screen
C1D4  CA                   DEX
C1D5  CA                   DEX
C1D6  10  E9               BPL   LOOK       ; go back one more time
C1D8  60                   RTS
C1D9  00          LOSER    .BYTE 0          ; this will hold a 0 or a 2
```

See also FIREBT, JOY2TO, JOYSTK.

Name

Read the two joysticks together as one stick

Description

With this routine in your programs, the user needn't worry about which joystick to use. **JOY2TO** combines the responses from both joysticks, handling the result as if it were coming from one stick.

The routine returns directional information on a character that's moved around the screen by POKEing. At the same time, it returns the status of the joystick fire buttons.

Prototype

1. AND the contents of the two joystick data registers together.
2. After performing an LSR, check the carry flag.
3. If carry is clear, decrement the row position for the character, provided you haven't reached the upper limit of the screen, and return to the main program. If the upper limit has been reached, simply exit the routine.
4. If carry was set in step 2, it indicates that neither joystick was moved in an upward direction. Repeat step 2 to check for downward, left, and right movement.
5. Check the fire buttons for both joysticks. If the fire-button bit (bit 4) is set, exit the routine.
6. Otherwise, store a zero to a fire-button flag (FIREFL) and RTS to the main program.

Explanation

Using **JOY2TO**, the program below draws with either joystick 1 or 2. By moving the joysticks in one of four directions, the ball character (SCCODE) "moves" across screen memory. Pressing a fire button clears the screen while the E key exits the program.

After initializing the row and column position of the ball, the corresponding screen memory location is calculated from $C017–$C04A. This series of instructions determines the screen position (SP) using the expression SP = (ROW * 40 + COLUMN) + 1024.

In order to multiply by numbers that aren't a power of 2 in machine language, such as 40, you have to break the multiplier down. In this case, first multiply the row by 4, then add the row once to this result: This is the same as multiplying the number by 5. Then multiply this by 8 (or 2↑3).

To multiply by 5, a single byte will suffice for the result. The screen row is never more than 24, so only a single byte is needed up to this point (5 * 24 = 120). But when you multiply this number by 8, since the result can exceed 255, two bytes are needed.

Once the screen location for the ball has been calculated and stored in zero page (ZP), the corresponding color memory location is determined and placed in ZP+3.

Following this is a delay of two jiffies. If this weren't included, joystick movement would be too rapid. If you add other routines to this code, a delay of one jiffy may be more suitable. But if you can't produce the effect you want, you may have to switch to a delay routine with more flexibility like **BYT2DL**.

Notice that within **JOYTO2**, we check the fire button at the end of the routine (in FIRE). In this case, we report its current status to the main program with the flag (FIREFL). When FIREFL is zero, a fire button is being pressed.

Routine

```
C000                SCREEN   =    1024        ; starting screen location
C000                ZP       =    251
C000                SCCODE   =    81          ; screen code for ball character
C000                COLVAL   =    3           ; color cyan
C000                TOPLIM   =    0           ; top row of screen
C000                LEFLIM   =    0           ; first column on left
C000                BOTLIM   =    24          ; bottom row of screen
C000                RIGLIM   =    39          ; last column on right
C000                XSTPOS   =    19          ; column 20 starting position
C000                YSTPOS   =    11          ; row 12 starting position
C000                CHROUT   =    65490
C000                CIAPRA   =    56320       ; data-port register A
C000                JIFFLO   =    162         ; low byte of jiffy clock
C000                NDX      =    198         ; NDX = 208 on the 128—number of
                                              ; characters in keyboard buffer
C000                LSTX     =    197         ; LSTX = 213 on the 128—matrix coordinate
                                              ; for last key pressed
                                              ;
                                              ; Draw with joystick 1 or 2. Clear screen with
                                              ; fire button. Quit on E key.
C000 A9 93          CLRCHR   LDA  #147        ; clear the screen
C002 20 D2 FF                JSR  CHROUT
C005 A9 13                   LDA  #XSTPOS     ; initialize starting position, column
C007 8D C3 C0                STA  XPOS
C00A A9 0B                   LDA  #YSTPOS     ; and row
C00C 8D C4 C0                STA  YPOS
C00F A9 01                   LDA  #1          ; also clear flag for fire buttons
C011 8D C5 C0                STA  FIREFL
C014 AD C4 C0       MOVE     LDA  YPOS        ; get row number
                                              ; And multiply it by 40.
C017 85 FB                   STA  ZP          ; save row temporarily
C019 0A                      ASL              ; multiply row by 4
C01A 0A                      ASL
C01B 65 FB                   ADC  ZP          ; add to row (carry cleared here by last ASL)
```

```
C01D  85  FB             STA   ZP          ; store result
                                           ; Multiply ZP by 8 (two-byte multiplication).
C01F  A9  00             LDA   #0          ; clear high byte of ZP
C021  85  FC             STA   ZP+1
C023  06  FB             ASL   ZP          ; double ZP, low byte first
C025  26  FC             ROL   ZP+1        ; then high byte
C027  06  FB             ASL   ZP          ; double ZP two more times
C029  26  FC             ROL   ZP+1
C02B  06  FB             ASL   ZP
C02D  26  FC             ROL   ZP+1
C02F  A5  FB             LDA   ZP          ; now add column number
C031  6D  C3  C0         ADC   XPOS        ; (carry cleared by last ROL ZP+1)
C034  85  FB             STA   ZP          ; store low byte of result
C036  A9  00             LDA   #0          ; add in carry to high byte
C038  65  FC             ADC   ZP+1
C03A  85  FC             STA   ZP+1        ; and store high byte
                                           ; Add in start of the screen.
C03C  18                 CLC               ; for addition
C03D  A9  00             LDA   #<SCREEN    ; get low byte of screen offset
C03F  65  FB             ADC   ZP          ; add in current position, low byte
C041  85  FB             STA   ZP          ; store low-byte result for screen position
C043  85  FD             STA   ZP+2        ; it's also the low byte result for color RAM
                                           ; position
C045  A9  04             LDA   #>SCREEN    ; get high byte of screen offset
C047  65  FC             ADC   ZP+1        ; add in high byte of position
C049  85  FC             STA   ZP+1        ; and store high-byte result
C04B  49  DC             EOR   #$DC        ; effectively add $D4 for high-byte color
                                           ; RAM offset
C04D  85  FE             STA   ZP+3        ; store high-byte result in zero page
C04F  A0  00             LDY   #0          ; as an index
C051  A9  03             LDA   #COLVAL     ; get the character color
C053  91  FD             STA   (ZP+2),Y    ; store color for ball in color RAM
C055  A9  51             LDA   #SCCODE     ; get the screen code
C057  91  FB             STA   (ZP),Y      ; store the ball to the screen
C059  A9  02             LDA   #2          ; for delay of two jiffies
C05B  65  A2             ADC   JIFFLO      ; add two to low byte of jiffy clock
C05D  C5  A2    DELAY    CMP   JIFFLO      ; wait for two jiffies
C05F  D0  FC             BNE   DELAY
C061  20  74  C0         JSR   JOY2TO      ; check both joysticks
C064  AD  C5  C0         LDA   FIREFL      ; check fire buttons
C067  F0  97             BEQ   CLRCHR      ; if either fire button pressed, clear the screen
C069  A9  00    BUFCLR   LDA   #0          ; clear keyboard buffer
C06B  85  C6             STA   NDX
C06D  A5  C5    MATGET   LDA   LSTX        ; get last key pressed
C06F  C9  0E             CMP   #14         ; is it E for exit?
C071  D0  A1             BNE   MOVE        ; if not E, then go to MOVE
C073  60        EXIT     RTS               ; if E pressed, then exit the program
                                           ;
                                           ; Total joystick conditions.
C074  AD  01  DC JOY2TO  LDA   CIAPRA+1    ; read joystick 1
C077  2D  00  DC         AND   CIAPRA      ; AND in joystick 2 reading
C07A  4A        UP       LSR               ; check up move
C07B  B0  0D             BCS   DOWN        ; not up
C07D  AD  C4  C0         LDA   YPOS        ; handle up, get row
C080  C9  00             CMP   #TOPLIM     ; compare to the top
C082  F0  3E             BEQ   EXITJS      ; top limit reached
C084  CE  C4  C0         DEC   YPOS        ; move up 1
C087  4C  C2  C0         JMP   EXITJS      ; and leave
C08A  4A        DOWN     LSR               ; check down move
C08B  B0  0D             BCS   LEFT        ; not down
C08D  AD  C4  C0         LDA   YPOS        ; handle down, get row
C090  C9  18             CMP   #BOTLIM     ; compare to screen bottom
C092  F0  2E             BEQ   EXITJS      ; bottom limit reached
C094  EE  C4  C0         INC   YPOS        ; move down 1
```

```
C097  4C C2 C0          JMP   EXITJS     ; and leave
C09A  4A         LEFT   LSR              ; check left move
C09B  B0 0D             BCS   RIGHT      ; not left
C09D  AD C3 C0          LDA   XPOS       ; handle left, get column
C0A0  C9 00             CMP   #LEFLIM    ; compare to left limit
C0A2  F0 1E             BEQ   EXITJS     ; left limit reached
C0A4  CE C3 C0          DEC   XPOS       ; move left 1
C0A7  4C C2 C0          JMP   EXITJS     ; and leave
C0AA  4A         RIGHT  LSR              ; check right move
C0AB  B0 0D             BCS   FIRE       ; not right
C0AD  AD C3 C0          LDA   XPOS       ; handle right, get column
C0B0  C9 27             CMP   #RIGLIM    ; compare to right limit
C0B2  F0 0E             BEQ   EXITJS     ; right limit reached
C0B4  EE C3 C0          INC   XPOS       ; move right 1
C0B7  4C C2 C0          JMP   EXITJS     ; and leave
C0BA  4A         FIRE   LSR              ; check fire buttons
C0BB  B0 05             BCS   EXITJS     ; not up, down, left, right, or fire
C0BD  A9 00             LDA   #0         ; a fire button pressed, so set flag
C0BF  8D C5 C0          STA   FIREFL
C0C2  60         EXITJS RTS              ; we're finished
                                         ;
                                         ; Starting positions follow.
C0C3  00         XPOS   .BYTE 0          ; current column number
C0C4  00         YPOS   .BYTE 0          ; current row
C0C5  01         FIREFL .BYTE 1          ; neither fire button pushed if equal to one,
                                         ; pushed if zero
```

See also FIREBT, JOY2SE, JOYSTK.

Name

Read a joystick

Description

You can add this routine to a program whenever you need to move a character about the screen with one of the joysticks. Before calling **JOYSTK**, define the border limits for the character in the equates and load the accumulator with the joystick number (1 or 2).

The routines return directional information as well as the status of the joystick fire button.

Prototype

1. Read the contents of the appropriate joystick data register into the accumulator.
2. After performing an LSR, check the carry flag.
3. If carry is clear, decrement the row position for the character, provided that you haven't reached the upper limit of the screen, and return to the main program. If the upper limit has been reached, simply exit the routine.
4. If carry is set in step 2, it indicates that the joystick is not moved in an upward direction. Repeat step 2 to check for downward, then left, and then right movement.
5. Finally, check the fire button bit. If it is set, exit the routine.
6. Otherwise, store a zero to a fire button flag (FIREFL) and RTS to the main program.

Explanation

The example program is almost identical to the program found under **JOY2TO**. Likewise, the two joystick routines themselves are quite similar.

The **JOY2TO** program POKEs the character moved around the screen along with its color byte. This one prints it with CHROUT after it has been positioned with **PLOTCR**. **TXTCOL** is used to color it. In the example program, the character moved by joystick 2 is the checked block—CHR$(166).

Since printing to the last screen position causes the screen to scroll, we limit the row position here to the first 24 rows (0–23).

The status of the fire button is returned to the calling program by using the flag FIREFL. FIREFL is zero when the fire button is being held down; otherwise, it's one.

Note: In using PLOT, remember that the row position loads into .X and the column into .Y. Also, be sure to clear the carry flag before you JSR to PLOT.

Routine

```
C000                CHAR     =    166          ; checkered block character
C000                COLVAL   =    4            ; color purple
C000                TOPLIM   =    0            ; top row of screen
C000                LEFLIM   =    0            ; first column on left
C000                BOTLIM   =    23           ; one row up from bottom of screen
C000                RIGLIM   =    39           ; last column on right
C000                XSTPOS   =    19           ; column 20 (starting position)
C000                YSTPOS   =    11           ; row 12 (starting position)
C000                CHROUT   =    65490
C000                PLOT     =    65520
C000                CIAPRA   =    56320        ; data port register A
C000                JIFFLO   =    162          ; low byte of jiffy clock
C000                NDX      =    198          ; NDX = 208 on the 128—number of
                                               , characters in keyboard buffer
C000                LSTX     =    197          ; LSTX = 213 on the 128—matrix coordinate
                                               ; for last key pressed
C000                COLOR    =    646          , COLOR = 241 on the 128—current text
                                               ; foreground color
                                               ;
                                               ; Draw with joystick 2. Clear screen when
                                               ; fire button pressed. Quit on E key.
C000  A9 93         CLRCHR   LDA  #147         ; clear the screen
C002  20 D2 FF               JSR  CHROUT
C005  A9 13                  LDA  #XSTPOS      , initialize starting position, column,
C007  8D 94 C0               STA  XPOS
C00A  A9 0B                  LDA  #YSTPOS      ; and row
C00C  8D 95 C0               STA  YPOS
C00F  A9 01                  LDA  #1           ; also clear fire button flag
C011  8D 96 C0               STA  FIREFL
C014  A9 04                  LDA  #COLVAL      ; store cursor color value
C016  8D 86 02      TXTCOL   STA  COLOR
C019  AC 94 C0      MOVE     LDY  XPOS         ; column position
C01C  AE 95 C0               LDX  YPOS         ; row position
C01F  18            PLOTCR   CLC               ; position the cursor at (.Y,.X)
C020  20 F0 FF               JSR  PLOT         ; position cursor
C023  A9 A6                  LDA  #CHAR        ; get the character to print
C025  20 D2 FF               JSR  CHROUT       , and print it
C028  A9 02                  LDA  #2           ; for delay of two jiffies
C02A  65 A2                  ADC  JIFFLO       ; add 2 to low byte of jiffy clock
C02C  C5 A2         DELAY    CMP  JIFFLO       , wait for two jiffies
C02E  D0 FC                  BNE  DELAY
C030  A9 02                  LDA  #2           ; joystick number
C032  20 45 C0               JSR  JOYSTK       , read joystick 2
C035  AD 96 C0               LDA  FIREFL       ; check fire button
C038  F0 C6                  BEQ  CLRCHR       ; if fire button pressed, clear the screen
C03A  A9 00         BUFCLR   LDA  #0           ; clear the keyboard buffer (if joystick 1 is
                                               ; used)
C03C  85 C6                  STA  NDX
C03E  A5 C5         MATGET   LDA  LSTX         ; get last key pressed
C040  C9 0E                  CMP  #14          ; is it E for exit?
C042  D0 D5                  BNE  MOVE         ; if not E, go to MOVE.
C044  60            EXIT     RTS               ; if E pressed, exit the program
                                               ;
                                               ; Enter with the joystick number in .A.
C045  29 01         JOYSTK   AND  #1           ; determine joystick offset
C047  AA                     TAX               ; put offset in .X
[redacted]  BD 00 DC         LDA  CIAPRA,X     ; read joystick 1 (.X = 1) or 2 (.X = 0)
```

```
C04B  4A              UP      LSR            ; check up move
C04C  B0  0D                  BCS   DOWN     ; not up
C04E  AD  95  C0              LDA   YPOS     ; handle up, get row
C051  C9  00                  CMP   #TOPLIM  ; compare to the top
C053  F0  3E                  BEQ   EXITJS   ; top limit reached
C055  CE  95  C0              DEC   YPOS     ; move up one
C058  4C  93  C0              JMP   EXITJS   ; and leave
C05B  4A              DOWN    LSR            ; check down move
C05C  B0  0D                  BCS   LEFT     ; not down
C05E  AD  95  C0              LDA   YPOS     ; handle down, get row
C061  C9  17                  CMP   #BOTLIM  ; compare to screen bottom
C063  F0  2E                  BEQ   EXITJS   ; bottom limit reached
C065  EE  95  C0              INC   YPOS     ; move down one
C068  4C  93  C0              JMP   EXITJS   , and leave
C06B  4A              LEFT    LSR            ; check left move
C06C  B0  0D                  BCS   RIGHT    , not left
C06E  AD  94  C0              LDA   XPOS     ; handle left, get column
C071  C9  00                  CMP   #LEFLIM  ; compare to left limit
C073  F0  1E                  BEQ   EXITJS   ; left limit reached
C075  CE  94  C0              DEC   XPOS     ; move left one
C078  4C  93  C0              JMP   EXITJS   ; and leave
C07B  4A              RIGHT   LSR            ; check right move
C07C  B0  0D                  BCS   FIRE     ; not right
C07E  AD  94  C0              LDA   XPOS     ; handle right, get column
C081  C9  27                  CMP   #RIGLIM  ; compare to right limit
C083  F0  0E                  BEQ   EXITJS   ; right limit reached
C085  EE  94  C0              INC   XPOS     ; move right one
C088  4C  93  C0              JMP   EXITJS   ; and leave
C08B  4A              FIRE    LSR            ; check fire button
C08C  B0  05                  BCS   EXITJS   ; not up, down, left, right, or fire
C08E  A9  00                  LDA   #0       ; fire button pressed, so set flag
C090  8D  96  C0              STA   FIREFL
C093  60              EXITJS  RTS            ; we're finished
                                             ;
C094  00              XPOS    .BYTE 0        ; current column position
C095  00              YPOS    .BYTE 0        , current row
C096  01              FIREFL  .BYTE 1        ; fire button not pushed if equal to 1, pushed
                                             ; if 0
```

See also FIREBT, JOY2TO, JOY2SE.

Name

Wait for a keypress

Description

KEYDEL causes a program to pause until a key is pressed.

Prototype

1. Clear the keyboard buffer by storing a zero in NDX.
2. Repeatedly JSR GETIN until the accumulator contains a nonzero value, indicating a key has been pressed.
3. When this happens, return to the main program.

Explanation

This routine is quite simple. **KEYDEL** clears the keyboard buffer and then, using the Kernal routine GETIN, fetches a keypress.

In the example program, we clear the screen, print a message, and then call **KEYDEL**. Pressing a key allows the program to continue. At this point, the screen is cleared again.

Note: If you need to know the actual key that was pressed while in **KEYDEL**, the accumulator will contain its ASCII value upon returning from the routine.

Routine

```
C000                NDX     =     198          ; NDX = 208 on the 128—number of
                                               ; characters in keyboard buffer
C000                GETIN   =     65508
C000                PLOT    =     65520
C000                CHROUT  =     65490
                                               ;
                                               ; Print a message and wait for a response.
                                               ; Then clear the screen.
C000  20 28 C0              JSR   CLRCHR       ; clear the screen
C003  A2 17                 LDX   #23          ; twenty-fourth row
C005  A0 07                 LDY   #7           ; eighth column
C007  18            PLOTCR  CLC                ; to position cursor at (7,23)
C008  20 F0 FF              JSR   PLOT         ; position cursor
C00B  A0 00                 LDY   #0           ; as an index in PRTLOP
C00D  B9 2D C0      PRTLOP  LDA   MSGSTR,Y     ; get a character from the message string
C010  F0 06                 BEQ   PRTEND       ; quit printing on zero byte
C012  20 D2 FF              JSR   CHROUT       ; and print it
C015  C8                    INY                ; for next character
C016  D0 F5                 BNE   PRTLOP       ; and continue printing
C018  20 1E C0      PRTEND  JSR   KEYDEL       ; wait for a keypress
C01B  4C 28 C0              JMP   CLRCHR       ; clear the screen and RTS
                                               ;
                                               ; Clear the keyboard buffer and wait for a
                                               ; keypress.
```

```
C01E  A9 00      KEYDEL  LDA   #0          ; clear the keyboard buffer (see BUFCLR)
C020  85 C6              STA   NDX
C022  20 E4 FF   KEYLOP  JSR   GETIN       ; get a keypress
C025  F0 FB              BEQ   KEYLOP      ; if no keypress
C027  60                 RTS               ; we've got a key
                                           ;
C028  A9 93      CLRCHR  LDA   #147        ; clear the screen
C02A  4C D2 FF           JMP   CHROUT      ; and RTS
                                           ;
C02D  50 52 45   MSGSTR  .ASC  "PRESS ANY KEY TO CONTINUE"
C046  00                 .BYTE 0           ; terminator byte
```

See also BYT1DL, BYT2DL, INTDEL, JIFDEL, TOD1DL.

Name

Load a program (ML or BASIC) to the location from which it was saved

Description

LOADAB performs an absolute load of an ML or BASIC program from disk. Thus, a program will be loaded into memory at the same address from which you saved it. If you wish to relocate the program as you load it, use **LOADBS** or **LOADRL**.

Prototype

1. On the 128, set the bank to 15.
2. Set up the parameters as 1,8,1 for an absolute load of the file (SETLFS, SETNAM).
3. On the 128, call SETBNK to specify the bank where the program is to be loaded and the bank containing its filename.
4. Load .A with zero to specify a load.
5. JSR to the Kernal LOAD routine.
6. If the program being loaded is in BASIC, store .X and .Y in the end-of-BASIC text pointer (VARTAB on the 64, TEXTTP on the 128).

Explanation

This routine, as written, relies on the file header information on the disk to load the program named PROGRAM. A secondary address of 1 causes the load to be absolute—that is, to the address specified in the program file itself.

Before calling the Kernal LOAD routine, place a zero in the accumulator. This tells the Kernal LOAD routine to load rather than to verify the program. Upon returning from LOAD, .X and .Y contain the low and high bytes, respectively, of the ending address of the file. For a 64 BASIC program, these should be placed in VARTAB, the two-byte end-of-BASIC text pointer at 45 (the equivalent pointer on the 128 is TEXTTP at 4624).

To use this routine to load your own BASIC programs, substitute for PROGRAM the name of the program you want to load. If you need to use the routine to load an ML program where it was saved, substitute the ML program name for PROGRAM. And since the program is not in BASIC, you can remove the STX VARTAB (STX TEXTTP on the 128) and STY VARTAB+1 (STY TEXTTP+1 on the 128) instructions following the JSR LOAD.

Note: **LOADAB** as presented lacks disk error checking. You can easily add this feature if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM as noted in the source listing. Jump to **DERRCK** immediately after the JSR LOAD instruction. Furthermore, be sure to open the error channel (15) at the beginning of the program (also noted in the source listing).

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000                SETLFS   =   65466
C000                SETNAM   =   65469
C000                LOAD     =   65493
C000                VARTAB   =   45         ; end-of-BASIC pointer—substitute
                                            ; TEXTTP = 4624 for the 128
C000                                        ; SETBNK = 65384; Kernal bank number for
                                            ; data and filename (128 only)
C000                                        ; MMUREG = 65280; MMU configuration
                                            ; register (128 only)
                                            ;
                                            ; Load BASIC (or ML) program into memory
                                            ; where it was saved.
                                            ;
                                            ; Open channel 15 here if you include error
                                            ; checking (DERRCK).
                                            ;
C000                LOADAB   =   *
                                            ; LDA #0; set bank 15 (128 only)
                                            ; STA MMUREG; (128 only)
C000  A9 01                  LDA  #1        ; logical file 1
C002  A2 08                  LDX  #8        ; device number for disk drive
C004  A0 01                  LDY  #1        ; secondary address of 1 causes absolute
                                            ; load
C006  20 BA FF               JSR  SETLFS    ; set for absolute load
                                            ; Include the following three instructions
                                            ; for the 128 only.
                                            ; LDA BNKNUM; bank for program
                                            ; LDX BNKFNM; bank containing filename
                                            ; JSR SETBNK
C009  A9 09                  LDA  #FNLENG   ; length of filename
C00B  A2 1C                  LDX  #<ILENM   ; address of filename
C00D  A0 C0                  LDY  #>ILENM
C00F  20 BD FF               JSR  SETNAM    ; set up filename
C012  A9 00                  LDA  #0        ; flag for load
C014  20 D5 FF               JSR  LOAD      ; load the file
                                            ;
                                            ; JSR DERRCK; insert for disk error
                                            ; checking
                                            ;
                                            ; For the 128, change VARTAB in next two
                                            ; instructions to TEXTTP.
C017  86 2D                  STX  VARTAB    ; store end-of-BASIC program address into
                                            ; pointer
```

```
C019  84  2E              STY   VARTAB+      ; (these two instructions can be deleted for
                                             ; ML program loads)
C01B  60                  RTS
                                             ;
                                             ; Insert DERRCK here if including error
                                             ; checking.
                                             ;
C01C  30  3A  50  FILENM  .ASC  "0,PROGRAM"; insert your filename here (<=16 characters)
C025              FNLENG  =     *-FILENM     ; length of filename
                                             ; Include the next two variables for the
                                             ; 128 only.
                                             ; BNKNUM  BYTE 0; bank number to load
                                             ; program into
                                             ; BNKFNM  BYTE 0; bank number where
                                             ; filename is located
```

See also LOADBS, LOADRL.

Name

Load a BASIC program into the current BASIC text area

Description

LOADBS performs a relative load of a BASIC program from disk. During this process, the load address in the file header on the disk is ignored. Instead, the program loads into the area of memory currently set aside for BASIC text.

If you want to relocate BASIC prior to loading the program, or if you need to load an ML program in this way, see **LOADRL**.

Prototype

1. On the 128, set the bank to 15.
2. Set up the parameters as 1,8,0 for a relative load of the file (SETLFS, SETNAM).
3. On the 128, call SETBNK to specify the bank in which to load the program and the bank containing the program filename.
4. Store zero in .A to specify a load.
5. Load .X and .Y with the starting address of BASIC from TXTTAB .
6. JSR to LOAD.
7. Store .X and .Y into the end-of-BASIC text pointer.
8. Relink the tokenized BASIC program text.

Explanation

This routine, as written, loads the BASIC program named "BASIC PROGRAM" into the BASIC text area. A secondary address of zero insures that the address in the file header will be overlooked when the program is positioned in memory.

Before JSRing to LOAD, the accumulator should be set to zero to load rather than to verify the file. The .X and .Y registers must contain the load address of the program. Since we're loading the program in the BASIC workspace, we can take this address from the two-byte pointer for the start-of-BASIC text area, TXTTAB.

Upon returning from the Kernal LOAD, .X and .Y contain the ending address of the program (plus 1). Complete the routine by storing these in the end-of-BASIC text pointer, VARTAB (TEXTTP for the 128), and relinking all program lines with the BASIC ROM routine LINKPG.

Note: **LOADBS** currently lacks disk error checking. You can add this feature if you like by incorporating the subroutine

DERRCK into the code. Place **DERRCK** just before FILENM as noted in the source listing. Jump to **DERRCK** immediately after the JSR LOAD instruction. Be sure to open the error channel (15) at the beginning of the program (also noted in the source listing).

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000              SETLFS    =    65466
C000              SETNAM    =    65469
C000              LOAD      =    65493
C000              TXTTAB    =    43            ; TXTTAB = 45 for the 128—start-of-BASIC
                                               ; pointer
C000              VARTAB    =    45            ; end-of-BASIC pointer—substitute
                                               ; TEXTTP = 4624 on the 128
C000              LINKPG    =    42291         ; LINKPG = 20303 for the 128
C000                                           ; SETBNK = 65384; Kernal bank number for
                                               ; data and filename (128 only)
C000                                           ; MMUREG = 65280, MMU configuration
                                               ; register (128 only)
                                               ;
                                               ; Load BASIC program into normal BASIC
                                               ; memory.
                                               ;
                                               ; Open channel 15 here if you include disk
                                               ; error checking (DERRCK).
                                               ;
C000              LOADBS    =    *
                                               ; LDA #0; set bank 15 (128 only)
                                               ; STA MMUREG; (128 only)
C000  A9 01                 LDA  #1            ; logical file 1
C002  A2 08                 LDX  #8            ; device number for disk drive
C004  A0 00                 LDY  #0            ; secondary address of zero causes relative
                                               ; load
C006  20 BA FF              JSR  SETLFS        ; set for relative load
                                               ; Include the following three instructions
                                               ; for the 128 only.
                                               ; LDA BNKNUM; bank for program
                                               ; LDX BNKFNM; bank containing filename
                                               ; JSR SETBNK
C009  A9 0F                 LDA  #FNLENG       ; length of filename
C00B  A2 23                 LDX  #<ILENM       ; address of filename
C00D  A0 C0                 LDY  #>ILENM
C00F  20 BD FF              JSR  SETNAM        ; set up filename
C012  A9 00                 LDA  #0            ; flag for load
C014  A6 2B                 LDX  TXTTAB        ; low byte of start-of-BASIC address
C016  A4 2C                 LDY  TXTTAB+       ; high byte of start-of-BASIC address
C018  20 D5 FF              JSR  LOAD          ; load program at the start of BASIC
                                               ;
                                               ; JSR DERRCK; insert for disk error
                                               ; checking
                                               ;
                                               ; For the 128, change VARTAB in next two
                                               ; instructions to TEXTTP.
C01B  86 2D                 STX  VARTAB        ; store end-of-BASIC program address into
                                               ; pointer
C01D  84 2E                 STY  VARTAB+
```

```
C01F  20  33  A5           JSR    LINKPG       ; relink lines of tokenized BASIC program
                                               ; text
C022  60                   RTS
                                               ;
                                               ; Insert DERRCK here if you are including
                                               ; error checking.
                                               ;
C023  30  3A  42  FILENM   .ASC   "0:BASIC PROGRAM"
                                               ; substitute your filename here (<=16
                                               ; characters)
C032              FNLENG   =      * FILENM     ; length of filename
                                               ; Include the next two variables for the
                                               ; 128 only
                                               ; BNKNUM .BYTE 0; bank number to which
                                               ; program is to be loaded
                                               ; BNKFNM .BYTE 0, bank number where
                                               ; filename is located
```

See also LOADAB, LOADRL.

Name

Load a BASIC or ML program at a designated memory address

Description

LOADRL is quite versatile. With it, you can load a BASIC or ML program from disk to any memory location specified. During this process, known as a *relative load,* the computer takes the load address from the X and Y registers rather than from the file (as is the case with absolute loads). Furthermore, if it's a BASIC program you're loading, **LOADRL** will even set the start-of-BASIC and end-of-BASIC pointers for you.

Prototype

1. On the 128, set the bank to 15.
2. Set up the parameters as 1,8,0 for a relocating load of the file (SETLFS, SETNAM).
3. On the 128, call SETBNK to specify the bank where the program is to be loaded and the bank containing its filename.
4. Store zero in .A to specify a load.
5. Store zero at the start-of-BASIC address (skip this step for ML loads).
6. Load .X and .Y with the load address (LOADAD).
7. Store this address in the start-of-BASIC pointer, TXTTAB (skip this step for ML loads).
8. JSR to the Kernal LOAD routine.
9. Store the contents of .X and .Y in the end-of-BASIC text pointer (skip this step for ML loads).
10. Relink the tokenized BASIC program text (skip this step for ML loads).

Explanation

The example routine is currently set up to load a BASIC program named "BASIC PROGRAM" at 16385 (LOADAD). To load your own BASIC program, just substitute its filename for "BASIC PROGRAM" and specify its load address as LOADAD in the equates. With the few additional changes given below, this same routine will just as easily perform an ML program load.

For all loads, whether BASIC or ML, a zero must be placed in the accumulator prior to JSRing to LOAD. This instructs the Kernal LOAD routine to load, rather than to verify, the program specified. If we're doing a BASIC program load, as in the example below, a zero must be placed in the byte

preceding the load address (or START, calculated in the equates). Since .A already contains a zero, we simply store this to START.

Furthermore, with a BASIC load, the start-of-BASIC text pointer (TXTTAB) must be set. Since the X and Y registers contain the load address (LOADAD) for the program prior to JSR LOAD, we can store these to TXTTAB at this time. This step is unnecessary with ML loads.

After executing the Kernal LOAD routine, you're finished if it's an ML program you're loading. But if you're doing a BASIC load (as in the example routine), you must store .X and .Y—which contain the ending address of the program (plus 1)—into VARTAB (the two-byte, end-of-BASIC text pointer) and relink all program lines with LINKPR. If you're working on a 128, change VARTAB to TEXTTP.

Note: **LOADRL** currently lacks disk error checking. You can easily add this if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM as noted in the source listing. Jump to **DERRCK** immediately after the JSR LOAD instruction. Be sure to open the error channel (15) at the beginning of the program, as noted in the listing.

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000          SETLFS   =   65466
C000          SETNAM   =   65469
C000          LOAD     =   65493
C000          LOADAD   =   16385     ; memory location where we want to put the
                                     ; program
C000          START    =   LOADAD-1  ; byte just prior to the start-of-BASIC text
C000          TXTTAB   =   43        ; TXTTAB = 45 on the 128—start-of-BASIC
                                     ; pointer
C000          VARTAB   =   45        ; end-of-BASIC pointer—substitute
                                     ; TEXTTP = 4624 for the 128
C000          LINKPG   =   42291     ; LINKPG = 20303 on the 128
C000                                 ; SETBNK = 65384; Kernal bank number for
                                     ; data and filename (128 only)
C000                                 ; MMUREG = 65280; MMU configuration
                                     ; register (128 only)
                                     ;
                                     ; Load the program "BASIC PROGRAM" at
                                     ; 16385.
                                     ;
                                     ; Open channel 15 here if you include disk
                                     ; error checking (DERRCK).
                                     ;
C000          LOADRL   =   *
                                     ; LDA #0; set bank 15 (128 only)
                                     ; STA MMUREG; (128 only)
```

```
C000  A9 01              LDA   #1           ; logical file 1
C002  A2 08              LDX   #8           ; device number for disk drive
C004  A0 00              LDY   #0           ; secondary address of zero causes
                                            ; relocating load
C006  20  BA  FF         JSR   SETLFS       ; set for relocating load
                                            ; Include the following three instructions
                                            ; on the 128 only.
                                            ; LDA BNKNUM; bank for program
                                            ; LDX BNKFNM; bank containing filename
                                            ; JSR SETBNK
C009  A9  0F             LDA   #FNLENG      ; length of filename
C00B  A2  2A             LDX   #<FILENM     ; address of filename
C00D  A0  C0             LDY   #>FILENM
C00F  20  BD  FF         JSR   SETNAM       ; set up filename
C012  A9  00             LDA   #0           ; flag for load
C014  8D  00  40         STA   START        ; store a zero at the start of BASIC (delete
                                            ; if loading ML)
C017  A2  01             LDX   #<LOADAD     ; set the load address
C019  86  2B             STX   TXTTAB       ; set start-of-BASIC pointer (delete if
                                            ; loading ML)
C01B  A0  40             LDY   #>LOADAD
C01D  84  2C             STY   TXTTAB+1     ; (also delete if loading ML)
C01F  20  D5  FF         JSR   LOAD         ; load the file at LOADAD
                                            ;
                                            ; JSR DERRCK; insert for disk error
                                            ; checking
                                            ;
                                            ; For the 128, change VARTAB in the next
                                            ; two instructions to TEXTTP.
C022  86  2D             STX   VARTAB       ; store end-of-BASIC program address into
                                            ; pointer
C024  84  2E             STY   VARTAB+1     ; (delete for ML loads)
C026  20  33  A5         JSR   LINKPG       ; relink lines of tokenized BASIC program
                                            ; text (delete if loading ML)
C029  60                 RTS
                                            ;
                                            ; Insert DERRCK here if you're including
                                            ; error checking.
                                            ;
C02A  30  3A  42  FILENM   .ASC  "0:BASIC PROGRAM"
                                            ; substitute your filename here (<=16
                                            ; characters)
C039              FNLENG   =     *-FILENM   ; length of filename
                                            ; Include the next two variables on the 128
                                            ; only
                                            ; BNKNUM .BYTE 0; bank number to which
                                            ; program is to be loaded
                                            ; BNKFNM .BYTE 0; bank number where
                                            ; ASCII filename is located
```

See also LOADAB, LOADBS.

Name

Get a character using the keyboard matrix

Description

At times you may want to get a keypress while ignoring the position of the shift keys (SHIFT, CTRL, and Commodore keys). For instance, suppose you wish to receive a yes/no (Y/N) response at some point in your program. If the user happens to have SHIFT LOCK down while responding, the input will be a graphics character. With **MATGET** this won't happen.

Prototype

1. Get the keyboard matrix value of the last key pressed from the register at 197 (213 on the 128).
2. Compare the value with the keycode for no key pressed (64 on the 64; 88 on the 128).
3. If no key has been pressed, get another value from the register.
4. Otherwise, compare the value in the register with the keycode for a specified key.
5. If this key has not been pressed, check the register again.

Explanation

This routine relies on memory location 197 (213 on the 128) to provide a keycode for the last key pressed. This location takes its values from the I/O register at 56321 during every normal interrupt.

The keycodes for each key on the 64 and 128 are given in the table. The first 64 (0–63) keycodes are identical on the two machines. Additional keycodes have been assigned to the extra keys on the 128, including the numeric keypad. This lets you distinguish between an upper-row number key and a numeric-keypad number key on this machine.

Keycodes for the 64 and 128

0 = INST/DEL	33 = I
1 = RETURN	34 = J
2 = CRSR right/left	35 = 0
3 = f7	36 = M
4 = f1	37 = K
5 = f3	38 = O
6 = f5	39 = N
7 = CRSR down/up	40 = +
8 = 3	41 = P
9 = W	42 = L
10 = A	43 = −
11 = 4	44 = .
12 = Z	45 = :
13 = S	46 = @
14 = E	47 = ,
15 = Not used	48 = £
16 = 5	49 = *
17 = R	50 = ;
18 = D	51 = CLR/HOME
19 = 6	52 = Not used
20 = C	53 = =
21 = F	54 = ↑
22 = T	55 = /
23 = X	56 = 1
24 = 7	57 = ←
25 = Y	58 = Not used
26 = G	59 = 2
27 = 8	60 = Space
28 = B	61 = Not used
29 = H	62 = Q
30 = U	63 = RUN/STOP
31 = V	64 = No key pressed (64) HELP (128)
32 = 9	

Additional 128 Keycodes

65 = 8 (keypad)	77 = 6 (keypad)
66 = 5 (keypad)	78 = 9 (keypad)
67 = TAB	79 = 3 (keypad)
68 = 2 (keypad)	80 = Not used
69 = 4 (keypad)	81 = 0 (keypad)
70 = 7 (keypad)	82 = . (keypad)
71 = 1 (keypad)	83 = ↑ (top)
72 = ESC	84 = ↓ (top)
73 = + (keypad)	85 = ← (top)
74 = − (keypad)	86 = → (top)
75 = LINE FEED	87 = NO SCROLL
76 = ENTER (keypad)	88 = No key pressed

In the example below, when an E has been pressed, we print it. This is to show that the E key has been pressed, either with or without any shift keys (SHIFT, CTRL, and Commodore keys) being held down.

Note: LSTX is updated during normal IRQ interrupts. If you write your own interrupt routine or perform an SEI to turn off interrupts, this routine will not work correctly (if at all). In such circumstances, you should call the Kernal routine SCNKEY (65439) to update LSTX before using this routine.

Routine

```
C000              LSTX     =     197         ; LSTX = 213 on the 128
C000              NOKEY    =     64          ; NOKEY = 88 on the 128
C000              CHROUT   =     65490

                                             ;
                                             ; Accept only E as input regardless of the
                                             ; positions of the shift keys
C000              MATGET   =     *
C000  A5 C5       WAIT     LDA   LSTX        ; get the last keypress
C002  C9 40                CMP   #NOKEY      ; compare to keycode for no key pressed
C004  F0 FA                BEQ   WAIT        ; if no keypress, then wait
C006  C9 0E                CMP   #14         ; keycode for E
C008  D0 F6                BNE   WAIT        ; no E, so get another keypress
C00A  A9 45                LDA   #69         ; character code for E (E key was pressed)
C00C  20 D2 FF             JSR   CHROUT      ; print it
C00F  60                   RTS
```

See also BUFCLR, CHRGTR, CHRGTS, CHRKER.

Name

Move BASIC text area above an ML program

Description

The 4K block of memory at 49152 on the 64 is the most popular area for storing machine language programs. If your program calls for more than one ML routine, you may be forced to position one of the routines elsewhere in memory.

Two alternative regions for locating ML routines are at the top or bottom of the BASIC text area. Assuming you choose one of these options, you often must protect your ML code from being overwritten by a coresident BASIC program. This particular routine shows how to position your ML programs below BASIC.

Prototype

1. Move the address of the end of the BASIC text area (in VARTAB) up by one page beyond the end of this program (**MBU64** plus your ML program). Also, change the pointer to the start of BASIC array space (ARYTAB) and the pointer to the top of string space (STREND) so they contain this address.
2. Store a zero to the low byte of TXTTAB to make the BASIC program start on an even-page boundary.
3. Store three zeros sequentially beginning at the address pointed to by TXTTAB.
4. Increment TXTTAB by 1.
5. Set the variable pointers (VARTAB, ARYTAB, and STREND) to point to an address two bytes beyond the start of BASIC text space (in TXTTAB) and return.

Explanation

To use **MBU64**, place your ML program at the end of this routine and then assemble both. The code at MLBAS ($801–$80C) provides you with a one-line BASIC program which SYSes to the start of **MBU64** at 2061. This line reads:

10 SYS2061

When you run this BASIC program and the SYS executes, **MBU64** moves the pointer to the start of BASIC program space (TXTTAB) above the end of your ML program (anywhere from 1 to 255 bytes above).

At the same time, several other BASIC pointers are altered, reflecting the fact that the BASIC program has been

NEWed. Among these are the end-of-BASIC pointer (VARTAB), the pointer to the start of BASIC array space (ARYTAB), and the pointer to the start of free RAM (STREND).

When moving BASIC up, remember that the location preceding the BASIC program text must contain a zero. For instance, suppose your BASIC program called for a hi-res screen at 8192. Since this screen occupies 8K of memory, you'll probably want to locate your BASIC program at 16384 or above.

If you choose to place it at 16384, a zero must be stored in this first location to mark the beginning of the BASIC program. TXTTAB, the start-of-BASIC pointer, in this instance, would point to 16385.

A BASIC program always ends with three zeros. The first one designates the end of the last program line, while the next two are the line link bytes. You can merge two BASIC programs by removing these last two zeros and storing a second BASIC program at this point in memory. If you attempt this, remember to relink the program lines for the two programs (by JSRing to LINKPRG at 42291) and adjust VARTAB, ARYTAB, and STREND to the end of the second program.

Routine

```
0801                TXTTAB   =     43          ; pointer to start of BASIC program
0801                VARTAB   =     45          ; pointer to end of BASIC program
0801                ARYTAB   =     47          ; pointer to start of BASIC array storage area
0801                STREND   =     49          ; pointer to end of string storage and start of
                                               ; free RAM
                                               ;
                                               ; Move the start of BASIC above your ML
                                               ; program. Program runs from BASIC
0801  0B 08         MLBAS    .BYTE 11,8        ; line link to 2059
0803  0A 00                  .BYTE 10,0        ; line number
0805  9E                     .BYTE 158         ; token for SYS
0806  32 30 36               ASC   "2061"      ; SYS address
080A  00 00 00               .BYTE 0,0,0       ; end of current line (0) and end of BASIC
                                               ; text (0,0)
                                               ;
                                               ; Move BASIC up.
080D  A6 2E         MBU64    LDX   VARTAB+1    ; load the high byte for the end of BASIC
                                               ; text
080F  E8                     INX               ; add one to move BASIC up by one page
                                               ; beyond this program
0810  86 2C                  STX   TXTTAB+1    ; and reset all pointers to this address
0812  86 2E                  STX   VARTAB+1
0814  86 30                  STX   ARYTAB+1
0816  86 32                  STX   STREND+1
0818  A9 00                  LDA   #0
081A  85 2B                  STA   TXTTAB      ; Set low byte of TXTTAB so it points to
                                               ; $XX00 (start of BASIC
                                               ; is now 1-255 bytes beyond the end of this
                                               ; program).
```

```
081C  A0 02              LDY  #2            ; as an index in ZERLOP
081E  91 2B     ZERLOP   STA  (TXTTAB),Y    ; put three zeros in memory pointed to by
                                            ; TXTTAB
0820  88                 DEY
0821  10 FB              BPL  ZERLOP        ; do three zeros
0823  A2 01              LDX  #1            ; TXTTAB increases by one to $XX01
0825  86 2B              STX  TXTTAB        ; so address pointed to by TXTTAB, and on
                                            ; either side are zeros
0827  E8                 INX                ; increment .X twice since variables start
                                            ; two bytes beyond start of BASIC
0828  E8                 INX
0829  86 2D              STX  VARTAB        ; and reset low byte of all variable pointers
082B  86 2F              STX  ARYTAB
082D  86 31              STX  STREND
082F  60                 RTS                ; end of the routine to move BASIC up
                                            ;
                                            ; Put the ML routine you want below BASIC
                                            ; here.
```

See also MBU128.

Name

Move BASIC text area above an ML program on the 128

Description

If you're using many ML routines simultaneously on the 128, you may be forced to position one or more of these in the normal BASIC text area beginning at 7169. Of course, any ML routines placed in this area of memory must be protected from being overwritten by the BASIC program.

One solution is to move the BASIC text up. This is the approach used here. By altering the start-of-BASIC pointer, **MBU128** lets you insert your ML routines below a coresident BASIC program.

Prototype

Before entering the routine (specifically in MLBAS), set up a BASIC line that will jump to the beginning of **MBU128**. This line should read as follows:

BANK0:SYS(PEEK(45)+PEEK(46)+32)

Do not insert any extra spaces in this line.

1. Within **MLB128**, move the start of BASIC up by one page beyond the end of the current BASIC program.
2. Adjust the end-of-BASIC pointer (TEXTTP) to point to this address.
3. Store a zero to the low byte of TXTTAB (start-of-BASIC pointer) so that BASIC starts on an even-page boundary.
4. Store three zeros sequentially beginning at the address pointed to by TXTTAB.
5. Increment TXTTAB by one.
6. Store a 3 into the low byte of TEXTTP since the end of BASIC is two bytes beyond the start of BASIC (with no BASIC program in memory) and RTS.

Explanation

To use **MBU128**, place your ML program at the end of this routine and then assemble both. The code at MLBAS ($1C01–$1C1D) provides you with a one line BASIC program which SYSes to the start of **MBU128** at 7201 in bank 0. This line reads

10 BANK0:SYS(PEEK(45)+256*PEEK(46)+32)

If you've previously used the GRAPHIC command, BASIC will relocate to $4000. If this is the case, you'll need to

adjust the high byte for the line link (currently at $1C02) to 64.

When you run this BASIC program and the SYS executes, **MBU128** moves the start-of-BASIC pointer (TXTTAB) above the end of your ML program (anywhere from 1 to 255 bytes above). At the same time, the end-of-BASIC pointer in TEXTTP is adjusted to point two bytes beyond this. The start of BASIC moves up and, in the process, the one-line BASIC program is NEWed.

The memory location preceding the BASIC program text must contain a zero. For instance, suppose you moved the start of a BASIC program to the address 8192. You'd place a zero in 8192, and TXTTAB, or the start-of-BASIC pointer, would have to point to 8193.

A BASIC program always ends with three zeros. The first one designates the end of the last program line, while the next two are the line link bytes. You can merge two BASIC programs together by removing these last two zeros and storing a second BASIC program at this point. If you do this, be sure to relink the lines for the two programs by JSRing to LINKPRG at 20303 and point TEXTTP to the end of the newly merged program.

Routine

```
1C01                TXTTAB    =     45            ; start-of BASIC program pointer
1C01                TEXTTP    =     4624          ; end-of-BASIC program pointer
                                                  ;
                                                  ; Move the start of BASIC above your ML
                                                  ; program—program runs from BASIC.
1C01  1F  1C        MLBAS     .BYTE 31,28         ; line link to 7199
1C03  0A  00                  .BYTE 10,0          ; line number
1C05  FE  02                  .BYTE 254,2         ; two-byte token for BANK
1C07  30  3A                  .BYTE 48,58         ; zero and colon
1C09  9E  28  C2              .BYTE $9E,$28,$C2,$28,$34,$35,$29,$AA
                                                  ; SYS(PEEK(45) +
1C11  32  35  36              .BYTE $32,$35,$36,$AC,$C2,$28,$34,$36,$29,$AA
                                                  ; 256*PEEK(46) +
1C1B  33  32                  .BYTE 51,50         ; offset of 32 from start of BASIC text )
1C1D  29  00  00              .BYTE $29,0,0,0     ; and three zeros for end of BASIC text
                                                  ;
                                                  ; Move BASIC up.
1C21  AE  11  12    MBU128    LDX   TEXTTP+1      ; load the high byte for the end of BASIC
                                                  ; text
1C24  E8                      INX                 ; add one to move BASIC up by one page
                                                  ; beyond this program
1C25  86  2E                  STX   TXTTAB+1      ; now reset the start-
1C27  8E  11  12              STX   TEXTTP+1      ; and end-of-BASIC pointers to this address
1C2A  A9  00                  LDA   #0
1C2C  85  2D                  STA   TXTTAB        ; set low byte of TXTTAB so that it points
                                                  ; to $XX00 (start of BASIC
                                                  ; is now 1–255 bytes beyond the end of this
                                                  ; program)
1C2E  A0  02                  LDY   #2            ; as an index in ZERLOP
```

```
1C30  91  2D      ZERLOP  STA  (TXTTAB),Y   ; put three zeros in memory pointed to by
                                            ; TXTTAB
1C32  88                  DEY
1C33  10  FB              BPL  ZERLOP       ; do all three
1C35  A2  01              LDX  #1           ; TXTTAB increased by one to $XX01
1C37  86  2D              STX  TXTTAB       ; so address at TXTTAB and on either side
                                            ; contains a zero
1C39  A2  03              LDX  #3           ; end of BASIC text is two bytes beyond
                                            ; start of BASIC
1C3B  8E  10  12          STX  TEXTTP
1C3E  60                  RTS               ; end of the routine to move memory
                                            ;
                                            ; Put the ML routine you want below BASIC
                                            ; here.
```

See also MBU64.

Name

Tune player

Description

MELODY provides a general framework for playing music. By changing certain parameters within this routine, you can adapt it to play any number of simple tunes.

Prototype

1. Before entering this routine, set up a table of notes which index values from a two-byte frequency table (NOTES), a table containing the relative durations for each note in NOTES (NDURTB), and a table of the two-byte frequencies needed for the tune (FREQTB).
2. Set a note counter (NOTENM) to zero.
3. Clear the SID chip with **SIDCLR** and select the necessary SID chip parameters (volume, attack/decay, and sustain/release).
4. In a loop (NOTELP), load the frequency for each note and store it in the frequency registers for voice 1.
5. Select a waveform (sawtooth in the example) and gate it.
6. Load the note's duration and cause a delay based on it.
7. Start the release cycle by ungating the waveform.
8. Increment the note counter and determine if all notes have played. If so, RTS. Otherwise, continue NOTELP to play the next note.

Explanation

MELODY plays a song by picking out notes from a table containing two-byte frequencies (FREQTB). These frequency values are the same ones given in the table of standard notes in your programmer's reference guide.

Currently, the values in FREQTB represent all the notes from G-4 (6430) through A-5 (14335). Alter this table depending upon which notes are used in your song. For instance, if your song ranged from G-2 to F-3, the frequencies in FREQTB would run from 1607 to 2864.

In building FREQTB, you really only need to list the actual note frequencies used in your song. But it generally appears less confusing if you include the entire range in the song, as we've done here. Furthermore, if the notes used are many or are selected from a wide range, you might let **NOTETB** generate a complete note table (all eight octaves) for you.

In order to get notes from FREQTB, a second table of in-

dex numbers (NOTES) is required. Each note selected plays for a period of time based on a duration value given in yet another table, NDURTB. The actual duration of each note is the number taken from NDURTB times eight jiffies, or 8/60 second.

In the example below then, the first note in NOTES, or G-4 with a frequency of 6430, plays for 8 jiffies; the second note, a C-5 with a frequency of 8583 plays for 16 jiffies; and so on.

This song plays in voice 1 using a sawtooth waveform. But other voices or waveforms may be more suitable for the song you're playing. In addition, you may want to change the other SID chip parameters such as the volume level, or the attack/decay and sustain/release rates.

For each song played with this routine, you need to work out not only the relative time each note plays (in NDURTB), but also the overall tempo of the song. The number of jiffies specified in the delay loop at $C036 determines a song's tempo. You may need to adjust this number, currently 8, up or down before the song sounds right.

Routine

```
C000                 FRELO1   =    54272        ; starting address for the SID chip
C000                 FREHI1   =    54273        ; voice 1 high frequency
C000                 VCREG1   =    54276        ; voice 1 control register
C000                 ATDCY1   =    54277        ; voice 1 attack/decay register
C000                 SUREL1   =    54278        ; voice 1 sustain/release register
C000                 SIGVOL   =    54296        ; SID chip volume register
C000                 JIFFLO   =    162          ; low byte of jiffy clock
                                                ;
                                                ; Play song.
C000  A9 00          MELODY   LDA  #0
C002  8D AB C0                STA  NOTENM       ; set pointer to first note in table
C005  20 AC C0                JSR  SIDCLR       ; clear the SID chip
C008  A9 0F                   LDA  #15          ; set the volume to maximum
C00A  8D 18 D4                STA  SIGVOL
C00D  A9 1A                   LDA  #$1A         ; set attack/decay
C00F  8D 05 D4                STA  ATDCY1
C012  A9 1B                   LDA  #$1B         ; set sustain/release
C014  8D 06 D4                STA  SUREL1
C017  AE AB C0       NOTELP   LDX  NOTENM       ; get the note number
C01A  BD 51 C0                LDA  NOTES,X      ; get index for FREQTB
C01D  0A                      ASL               ; double it since FREQTB contains two-
                                                ; byte addresses
C01E  AA                      TAX               ; to index FREQTB
C01F  BD 8D C0                LDA  FREQTB,X     ; get low byte of note's frequency
C022  8D 00 D4                STA  FRELO1       ; store it in voice 1
C025  BD 8E C0                LDA  FREQTB+1,X   ; get high byte of note's frequency
C028  8D 01 D4                STA  FREHI1       ; store it in voice 1
C02B  A9 21                   LDA  #%00100001   ; gate sawtooth waveform
C02D  8D 04 D4                STA  VCREG1
C030  AE AB C0                LDX  NOTENM       ; put the note number in .X
C033  BC 6F C0                LDY  NDURTB,X     ; get the note's duration from a table
C036  A9 08          REPEAT   LDA  #8           ; delay for number of jiffies in .A
C038  65 A2                   ADC  JIFFLO
C03A  C5 A2          DELAY    CMP  JIFFLO       ; has the time elapsed?
```

```
C03C  D0 FC           BNE   DELAY         ; if not, continue the delay
C03E  88              DEY
C03F  D0 F5           BNE   REPEAT        ; repeat the jiffy delay if necessary
C041  A9 20           LDA   #%00100000    ; ungate waveform
C043  8D 04 D4        STA   VCREG1
C046  EE AB C0        INC   NOTENM        ; increase note counter
C049  AD AB C0        LDA   NOTENM
C04C  C9 1E           CMP   #NMNOTE       ; see if all notes have played
C04E  90 C7           BCC   NOTELP        ; if not, then continue
C050  60              RTS                 ; that's all
                                          ;
C051  00 05 05 NOTES  .BYTE 0,5,5,7,7,9,12,9,5,0,5,5,7,7,9,5
                                          ; table of notes
C061  00 05 05        .BYTE 0,5,5,7,7,9,12,9,5,14,7,10,9,5
C06F           NMNOTE =     * NOTES       ; number of notes
C06F  01 02 01 NDURTB .BYTE 1,2,1,2,1,1,1,1,2,1,2,1,2,1,3,3
                                          ; table of note durations
C07F  01 02 01        .BYTE 1,2,1,2,1,1,1,1,3,3,2,1,3,3
C08D  1E 19 9C FREQTB .WORD 6430,6812,7217,7647,8101,8583,9094
                                          ; table of 2-byte frequency values
C09B  A2 25 DF        .WORD 9634,10207,10814,11457,12139,12860,13625,14435
C0AB  00       NOTENM .BYTE 0             ; note number counter
                                          ;
                                          ; Clear the SID chip.
C0AC  A9 00    SIDCLR LDA   #0            ; fill with zeros
C0AE  A0 18           LDY   #24           ; as the offset from FRELO1
C0B0  99 00 D4 SIDLOP STA   FRELO1,Y      ; store zero in each SID chip address
C0B3  88              DEY                 ; for next lower address
C0B4  10 FA           BPL   SIDLOP        ; fill 25 bytes
C0B6  60              RTS                 ; we're done
```

See also BEEPER, BELLRG, EXPLOD, INTMUS, NOTETB, SIDCLR, SIDVOL, SIRENS.

Name

Change mixed-case characters to all lowercase

Description

MIXLOW takes a letter in the accumulator and returns it as lowercase in .A. The X and Y registers are unaffected by the routine. So, you can access **MIXLOW** from within a loop indexed by .X or .Y without needing to save and restore the index register.

In a word processor, this routine would be practical for setting up a search function. Let's say you want to find all occurrences of the word *computer,* whether the lettering is uppercase, lowercase, or a combination of the two. **MIXLOW** will help you with this process, converting each character of the specified word to lowercase. So, if *Computer* and *COMPUTER* appear in your document, both will be found.

Prototype

1. Determine whether the character in .A is less than the uppercase range.
2. If so, then RTS.
3. Determine whether the character is less than CHR$(123), putting it in the first uppercase range, 97–122.
4. If it is, subtract 32 to put it in the lowercase range and RTS.
5. If the character value exceeds 122, check to see whether it's in the second uppercase range of 193–218.
6. If it is, convert it to lowercase by ANDing with 127 and RTS.

Explanation

The example routine first switches in lowercase/uppercase mode. An ASCII string (STRING) in mixed case is read in. Each letter of the string is converted to lowercase and printed with CHROUT. Before exiting the routine, the SHIFT/ Commodore key combination is reenabled to allow case switching.

Note: When converting characters in the range 193–218 to lowercase, we AND with 127. This effectively subtracts 128, but saves a byte in the code (as opposed to using SEC: SBC #128).

Routine

```
C000          CHROUT  =   65490
C000          DSFTCM  =   8          ; DSFTCM = 11 on the 128
C000          ESFTCM  =   9          ; ESFTCM = 12 on the 128
                                     ;
```

```
                                        ; Convert an upper/lowercase string to all
                                        ; lowercase.
C000  A9 0E             LDA  #14        ; switch to lowercase/uppercase mode
C002  20 D2 FF          JSR  CHROUT
C005  A9 08             LDA  #DSFTCM    ; disable case switching with
                                        ; SHIFT/Commodore key
C007  20 D2 FF          JSR  CHROUT
                                        ; Print string as all lowercase.
C00A  A0 00             LDY  #0         ; as an index
C00C  B9 37 C0 LOOP     LDA  STRING,Y   ; get a character from string
C00F  F0 09             BEQ  FINISH     ; is it a zero byte?
C011  20 20 C0          JSR  MIXLOW     ; convert to lowercase
C014  20 D2 FF          JSR  CHROUT     ; print it
C017  C8                INY             ; next character
C018  D0 F2             BNE  LOOP       ; continue printing
C01A  A9 09    FINISH   LDA  #ESFTCM    ; enable SHIFT/Commodore key case
                                        ; switching
C01C  20 D2 FF          JSR  CHROUT
C01F  60                RTS
                                        ;
                                        ; Convert mixed case in .A to all lowercase.
                                        ; Return in .A.
C020  C9 61    MIXLOW   CMP  #97        ; is it less than uppercase A?
C022  90 12             BCC  EXIT       ; yes, so exit
C024  C9 7B             CMP  #123       ; is it greater than uppercase Z?
C026  B0 04             BCS  SECSET     ; yes, so check for second uppercase set
                                        ; Character is in ASCII range 97–122,
C028  38                SEC
C029  E9 20             SBC  #32        ; so subtract 32 to put it in range 65–90
C02B  60                RTS             ; and exit
C02C  C9 C1    SECSET   CMP  #193       ; is it less than second uppercase A?
C02E  90 06             BCC  EXIT       ; yes, so exit
C030  C9 DB             CMP  #219       ; is it greater than second uppercase Z?
C032  B0 02             BCS  EXIT       ; yes, so exit
                                        ; character is in ASCII range 193–218
C034  29 7F             AND  #127       ; so effectively subtract 128 to put in range
                                        ; 65–90
C036  60       EXIT     RTS
                                        ;
C037  C3 48 C1 STRING   .ASC "ChAnGe MiXeD cAsE tO aLl LoWeRcAsE"
C059  00                .BYTE0
```

See also CNVERT, MIXUPP, SWITCH.

Name

Convert mixed case characters to all uppercase

Description

MIXUPP takes the letter in the accumulator and returns it as uppercase in .A. In the process, .X and .Y are left intact. This routine is handy anytime you want only uppercase input—for instance, when filenames are requested or when a letter response is sought (Y/N).

Prototype

1. Determine whether the character in .A is in the lowercase range, 65–90.
2. If not, RTS.
3. Otherwise, add 32 to put it in the uppercase range, 97–122, and RTS.

Explanation

The example routine switches in the lowercase/uppercase character set, accepts individual characters with GETIN, converts them to uppercase with **MIXUPP**, and finally prints them with CHROUT. Pressing RETURN exits the routine. In the process, case switching with SHIFT/Commodore key is reenabled.

Note: A CLC is not required before 32 is added in **MIXUPP**. If the program falls through BCS, carry will already have been cleared.

Routine

```
C000                CHROUT  =    65490
C000                GETIN   =    65508
C000                DSFTCM  =    8           ; DSFTCM = 11 on the 128
C000                ESFTCM  =    9           ; ESFTCM = 12 on the 128
                                             ;
                                             ; Convert uppercase/lowercase input to all
                                             ; uppercase; quit on RETURN.
C000  A9 0E                 LDA  #14         ; switch to lowercase/uppercase mode
C002  20 D2 FF              JSR  CHROUT
C005  A9 08                 LDA  #DSFTCM     ; disable SHIFT/Commodore key case
                                             ; switching
C007  20 D2 FF              JSR  CHROUT
C00A  20 E4 FF      LOOP    JSR  GETIN       ; get a character
C00D  F0 FB                 BEQ  LOOP        ; if no input, wait
C00F  C9 0D                 CMP  #13         ; is it RETURN?
C011  F0 08                 BEQ  QUIT        ; yes, so leave
C013  20 21 C0              JSR  MIXUPP      ; convert to all uppercase
C016  20 D2 FF              JSR  CHROUT      ; and print it
C019  D0 EF                 BNE  LOOP        ; get another character
C01B  A9 09         QUIT    LDA  #ESFTCM     ; enable SHIFT/Commodore key case
                                             ; switching
```

```
C01D  20  D2  FF            JSR   CHROUT
C020  60                    RTS
                                          ;
                                          ; Convert ASCII input in .A to all uppercase.
                                          ; Return value in .A.
C021  C9  41       MIXUPP   CMP   #65     ; is it less than lowercase a?
C023  90  06                BCC   EXIT    ; yes, so exit
C025  C9  5B                CMP   #91     ; is it greater than lowercase z?
C027  B0  02                BCS   EXIT    ; yes, so exit
                                          ; Add 32 to put in ASCII range 97-122.
C029  69  20                ADC   #32     ; note that carry is already clear if we fall
                                          ; through prior instruction
C02B  60           EXIT     RTS
```

See also CNVERT, MIXLOW, SWITCH.

Name

Move a block of data downward in memory

Description

Specifically designed to move blocks of data down in memory, this routine can be used to move other machine language routines, or text and numeric data tables. Provided the source and destination blocks don't overlap, **MOVEDN** will also move memory up.

Prototype

In the initialization routine (MDINIT):

1. Store the two-byte origin address (here, BLOCK1) in ZP and the two-byte target, or destination, address (here, BLOCK2) in ZP+2.
2. Store the number of bytes to move down (NUMBER in the equates) in .X (low byte) and .Y (high byte).

In **MOVEDN**:

1. Store the number of bytes to move, currently in .X and .Y, into a two-byte counter (COUNTR).
2. Using indirect addressing in DOWNLP, transfer bytes from the source memory block (at ZP) to the target memory block (at ZP+2).
3. On the 128, you can move memory from one bank to another. Define BNKSRC (source bank number) and BNKTAR (target bank number) at the end of the program with the appropriate banks. Replace the LDA (ZP),Y at DOWNLP with the three instructions that follow it in the listing and the STA (ZP+2),Y just below this with the next four instructions (labeled 128 only).
4. Increase both zero-page pointers by one with the subroutine ADDONE.
5. Decrement the bytes counter (COUNTR), continuing DOWNLP until all bytes from the source block have been moved. Then RTS.

Explanation

The following program shows how **MOVEDN** might be used in a word processor to delete text from the screen.

After printing a message to the screen, the program waits for a keypress. If D is pressed, a portion of the message is deleted, and the program ends.

When you press D, the program calls the subroutine MDINIT, then **MOVEDN**. MDINIT tells **MOVEDN** where the source and target blocks begin (in ZP and ZP+2), and also how many bytes to move. Upon entering **MOVEDN**, the number of bytes to move is stored to a two-byte counter (COUNTR) which decrements during the memory transfer process. At the same time, the zero-page pointers to the source and target blocks are incremented. When COUNTR reaches zero, the transfer is complete.

Because it relies on zero-page addressing, on the 128, **MOVEDN** can be readily modified to move memory from bank to bank. To accomplish this, you need two Kernal routines: INDFET, which performs an indirect load into the accumulator from the bank in .X, and INDSTA, which stores .A indirectly into the bank in .X. To implement these routines, replace the LDA (ZP),Y at $C046 with the commented instructions that follow (DOWNLP LDA #ZP:LDX BNKSRC:JSR INDFET) and replace the STA (ZP+2),Y at $C048 with LDX #ZP+2:STX 697:LDX BNKTAR:JSR INDSTA. Also include the bank numbers for the source (BNKSRC) and target (BNKTAR) blocks, defined at the end of the program.

If you want to use **MOVEDN** to move memory up, before assembling the routine, switch the definitions of BLOCK1 and BLOCK2 so that BLOCK1 is lower in memory. Of course, in order for this method to succeed, the two memory blocks must not overlap.

Note: Unlike some memory move routines (such as **SWAPIT**), **MOVEDN** has no error checking. It's up to you to make sure the memory blocks you've defined in the equates are in the proper relative position in memory.

Routine

```
C000                ZP       =    251
C000                CHROUT   =    65490
C000                PLOT     =    65520
C000                GETIN    =    65508
C000                BLOCK1   =    1267       ; memory block 1 (source)
C000                BLOCK2   =    1262       ; memory block 2 (target)
C000                NUMBER   =    757        ; number of bytes to move down
C000                INDFET   =    65396      ; Kernal routine to load indirectly from any
                                             ; bank (128 only)
C000                INDSTA   =    65399      ; Kernal routine to store indirectly to any
                                             ; bank (128 only)
                                             ;
                                             ; Print a message to the screen. Delete a word
                                             ; on D
C000  A9 93         CLRCHR   LDA  #147       ; clear the screen
C002  20 D2 FF               JSR  CHROUT
```

```
C005  A2 05              LDX  #5             ; row number (sixth row)
C007  A0 1E              LDY  #30            ; column number (thirty-first column)
C009  18        PLOTCR   CLC                 , clear carry to set position
C00A  20 F0 FF           JSR  PLOT           ; position cursor at (.Y,.X)
C00D  A0 00              LDY  #0             ; as an index in PRTLOP
C00F  B9 6D C0  PRTLOP   LDA  TXTSTR,Y       ; get a character from TXTSTR
C012  F0 06              BEQ  GETKEY         ; exit PRTLOP on zero byte
C014  20 D2 FF           JSR  CHROUT         ; print the character
C017  C8                 INY                 ; for next character
C018  D0 F5              BNE  PRTLOP         ; branch always
C01A  20 E4 FF  GETKEY   JSR  GETIN          ; look for D
C01D  F0 FB              BEQ  GETKEY         ; if no keypress
C01F  C9 44              CMP  #68            ; is it D?
C021  D0 F7              BNE  GETKEY         , if not D, get another keypress
C023  20 2A C0           JSR  MDINIT         ; initialize zero-page pointers and get number
                                             ; of bytes to move
C026  20 3F C0           JSR  MOVEDN         ; move bytes down
C029  60                 RTS
                                             ;
                                             ; Initialize zero-page pointers to BLOCK1 and
                                             ; BLOCK2. Two bytes at ZP point
                                             , to source, and two at ZP+2 point to target.
                                             ; Also put NUMBER in .X and .Y.
C02A  A9 F3     MDINIT   LDA  #<BLOCK1       ; low byte of BLOCK1 first
C02C  85 FB              STA  ZP
C02E  A2 04              LDX  #>BLOCK1       ; then high byte
C030  86 FC              STX  ZP+1
C032  A9 EE              LDA  #<BLOCK2       ; and again for BLOCK2
C034  85 FD              STA  ZP+2
C036  A2 04              LDX  #>BLOCK2
C038  86 FE              STX  ZP+3
C03A  A2 F5              LDX  #<NUMBER       , then put low byte of number of bytes to
                                             ; move down in .X
C03C  A0 02              LDY  #>NUMBER       ; and high byte in .Y
C03E  60                 RTS
                                             ;
                                             ; Move bytes down. Enter with the number
                                             ; of bytes to move in .X (low)
                                             ; and .Y (high). Source block is in two bytes
                                             ; at ZP, and target block at ZP+2.
C03F  8E 6B C0  MOVEDN   STX  COUNTR         ; store number to COUNTR, low byte first
C042  8C 6C C0           STY  COUNTR+1       ; then high byte
C045  A0 00              LDY  #0             ; index for DOWNLP
C047  B1 FB     DOWNLP   LDA  (ZP),Y         ; get a byte from source block
                                             ;
                                             ; On the 128, substitute the next three lines
                                             ; for the previous line
                                             ; to move memory from bank to bank.
                                             ; DOWNLP LDA #ZP; put zero-page
                                             ; pointer to source block in .A
                                             ; LDX BNKSRC; bank number for source
                                             ; block
                                             ; JSR INDFET; load indirectly from bank .X
                                             ; beginning at source
                                             ;
C049  91 FD              STA  (ZP+2),Y       ; store it in target block
                                             ;
                                             ; Again, on the 128, substitute the next four
                                             ; lines for the previous line
                                             ; to move from bank to bank.
                                             ; LDX #ZP+2; put zero-page pointer to
                                             ; target block in 697
                                             ; STX 697
                                             ; LDX BNKTAR; bank number for target
```

```
                                             ; block
                                             ; JSR INDSTA; store indirectly from bank
                                             ; .X beginning at target
                                             ;
C04B  20 5E C0           JSR   ADDONE        ; increase ZP pointers by one
C04E  CE 6B C0           DEC   COUNTR        ; decrement counter low byte
C051  D0 F4              BNE   DOWNLP        ; if low byte hasn't turned over, continue
                                             ; moving memory down
C053  CE 6C C0           DEC   COUNTR+1      ; otherwise, decrement the high byte
C056  AD 6C C0           LDA   COUNTR+1      ; determine whether we've moved the last
                                             ; page
C059  C9 FF              CMP   #255          ; on the last page, high byte of counter goes
                                             ; from 0 through 255
C05B  D0 EA              BNE   DOWNLP        ; if not on the last page, continue
C05D  60                 RTS
                                             ;
                                             ; Increment zero-page pointers by one.
C05E  E6 FB     ADDONE   INC   ZP            ; increment low byte of source
C060  D0 02              BNE   INCTAR        ; if it hasn't turned over, handle target
                                             ; pointers
C062  E6 FC              INC   ZP+1          ; increment high byte of source block
C064  E6 FD     INCTAR   INC   ZP+2          ; do the same for target pointers
                                             ; increment low byte first
C066  D0 02              BNE   ADEXIT        ; if it hasn't turned over, exit ADDONE
C068  E6 FE              INC   ZP+3          ; and increment high byte, if necessary
C06A  60        ADEXIT   RTS
                                             ;
C06B  00 00     COUNTR   .WORD 0             ; two-byte counter for remaining number of
                                             ; bytes to move down
C06D  54 48 49  TXTSTR   .ASC  "THIS IS LINE 6 AND 7  DELETE 'LINE ' ON D."
C097  00                 .BYTE 0             ; terminator byte
                                             ; BNKSRC .BYTE 0; the bank number where
                                             ; source is (128 only)
                                             ; BNKTAR .BYTE 0; the bank number where
                                             ; target is (128 only)
```

See also MVU128, MVU64, SWAPIT.

Name

Move sprite to an absolute (predetermined) screen location

Description

In some situations—board games or menu programs, for example—you may want to position sprites at certain fixed locations. When the sprite moves, it doesn't glide smoothly from one spot to another; it jumps directly to the new place. This routine uses a lookup table to put a sprite into position.

Prototype

1. Enter the routine with .X holding low byte of the x coordinate, .A holding the high byte of the x coordinate (1 or 0), and .Y holding the y coordinate.
2. Store the values in the appropriate VIC registers.

Explanation

The framing program prints a numeric grid on the screen, with the numbers 1–9 in a 3 × 3 square. It checks for a keypress, and when any of the numbers 1–9 is pressed, a box-shaped sprite is moved to the appropriate position on the grid. Press the zero key to exit.

The **MOVSAB** routine is very simple—three lines plus an RTS. Most of the example program is spent setting up the screen and creating the sprite shape. Note the message at the bottom. The 17s and 157s are cursor-down and cursor-left characters used to print the screen grid.

Note: This routine moves only one sprite. If you want to handle several, you'll need an additional variable that indicates which sprite should be moved.

The 128's BASIC 7.0 has a variety of very useful commands for controlling sprites. Unfortunately, when you're trying to control sprites from ML, BASIC tends to get in the way. To disable the 128's various sprite commands, enter POKE 4861,1 (or any other non-zero value) before you SYS to this routine.

Routine

```
C000          SPCOLR   =   53287     ; sprite 0 color
C000          SPX      =   53248     ; x position
C000          SPY      =   53249     ; y position
C000          SPXM     =   53264     ; MSB bit of x position
C000          SPE      =   53269     ; sprite enable
C000          SPP      =   2040      ; pointer to sprite zero
C000          SPSHAPE  =   832       ; SPSHAPE = 3584 on the 128—address of
                                     ; shape data
C000          POINTR   =   13        ; POINTR = 56 on the 128 (56*64)—pointer
```

```
                                               ; to shape data
C000                  PLOT     =     $FFF0     ; Kernal plot routine
C000                  CHROUT   =     $FFD2     ; Kernal print routine
C000                  GETIN    =     $FFE4     ; get a key
C000  20  3F  C0               JSR   SETSPR    ; set up sprite
C003  20  78  C0               JSR   SCREEN    ; print numbers 1-9 on screen
C006  20  93  C0      MAIN     JSR   GETKEY    ; get a key 1-9—the number 1-9 is in .X
C009  E0  00                   CPX   #0        ; is it a zero?
C00B  D0  01                   BNE   MOVEIT    ; no, move the sprite
C00D  60                       RTS             ; yes, quit this program
C00E  CA              MOVEIT   DEX             ; subtract one, so it works right
C00F  BD  2D  C0               LDA   XLO,X     ; get the low byte of the x position
C012  8D  CC  C0               STA   TEMP      ; save it temporarily
C015  BC  36  C0               LDY   YLO,X     ; get the y position
C018  BD  24  C0               LDA   XHI,X     ; and the high byte of x
C01B  AE  CC  C0               LDX   TEMP      ; now the real x position
C01E  20  A4  C0               JSR   MOVSAB    ; call the move absolute routine
C021  4C  06  C0               JMP   MAIN      ; go back for more
C024  00  01  01      XHI      .BYTE 0,1,1,0,1,1,0,1,1
C02D  F6  06  16      XLO      .BYTE 246,6,22,246,6,22,246,6,22
C036  40  40  40      YLO      .BYTE 64,64,64,80,80,80,96,96,96
                                               ;
C03F  A9  01          SETSPR   LDA   #%00000001 ; turn on sprite 0
C041  8D  15  D0               STA   SPE       ; setting bit 0 in sprite-enable
C044  A9  07                   LDA   #7        ; color yellow
C046  8D  27  D0               STA   SPCOLR    ; into the color register
C049  A9  00                   LDA   #0        ; position zero
C04B  8D  00  D0               STA   SPX       ; in x low byte
C04E  8D  10  D0               STA   SPXM      ; in x high byte
C051  8D  01  D0               STA   SPY       ; and y location
                                               ;
C054  A9  00                   LDA   #0        ; zero to clear out the shape
C056  A0  40                   LDY   #64
C058  99  40  03      CLSP     STA   SPSHAPE,Y ; clear it out
C05B  88                       DEY
C05C  10  FA                   BPL   CLSP      ; all 63 bytes
                                               ;
C05E  A2  0A                   LDX   #10       ; ten lines
C060  A0  00                   LDY   #0        ; start at zero
C062  A9  FF          CREATE   LDA   #255
C064  99  40  03               STA   SPSHAPE,Y
C067  C8                       INY
C068  A9  C0                   LDA   #192
C06A  99  40  03               STA   SPSHAPE,Y
C06D  C8                       INY
C06E  C8                       INY
C06F  CA                       DEX
C070  D0  F0                   BNE   CREATE
C072  A9  0D                   LDA   #POINTR   ; set the pointer
C074  8D  F8  07               STA   SPP
C077  60                       RTS
                                               ;
C078  A9  93          SCREEN   LDA   #147      ; clear screen character
C07A  20  D2  FF               JSR   CHROUT    ; print it
                                               ;
C07D  A0  1C                   LDY   #28       ; getting ready to plot—twenty-eighth
                                               ; column
C07F  A2  02                   LDX   #2        ; second row
C081  18                       CLC             ; clear carry to plot
C082  20  F0  FF               JSR   PLOT      ; now the cursor is ready
C085  A0  00                   LDY   #0        ; print the screen
C087  B9  AE  C0      PLOOP    LDA   MESSAGE,Y
C08A  F0  06                   BEQ   QPLP      ; if it's zero, quit
C08C  20  D2  FF               JSR   CHROUT    ; else print it
```

```
C08F  C8                     INY
C090  D0  F5                 BNE   PLOOP      ; (branch always)
C092  60           QPLP      RTS
                                              ;
C093  20  E4  FF   GETKEY    JSR   GETIN      ; get a key
C096  F0  FB                 BEQ   GETKEY     ; no key pressed, go back
C098  C9  30                 CMP   #48        ; lower than ASCII 0?
C09A  90  F7                 BCC   GETKEY     ; yes, go back
C09C  C9  3A                 CMP   #58        ; higher than ASCII 9?
C09E  B0  F3                 BCS   GETKEY     ; yes, try again
C0A0  29  0F                 AND   #15        ; strip off the extra stuff
C0A2  AA                     TAX              ; and transfer from A to X
C0A3  60                     RTS              ; we're done here
                                              ;
C0A4               MOVSAB    =     *          ; the main routine
C0A4  8D  10  D0             STA   SPXM       ; most significant bit
C0A7  8E  00  D0             STX   SPX        ; the x position
C0AA  8C  01  D0             STY   SPY        ; the y position
C0AD  60                     RTS              ; all done
                                              ;
C0AE  31  20  32   MESSAGE   .ASC  "1 2 3"
C0B3  11  11  9D             .BYTE 17,17,157,157,157,157,157
C0BA  34  20  35             .ASC  "4 5 6"
C0BF  11  11  9D             .BYTE 17,17,157,157 157,157,157
C0C6  37  20  38             .ASC  "7 8 9"
C0CB  00                     .BYTE 0
C0CC  00           TEMP      .BYTE 0
```

See also SPRINT.

Name

Set the colors for multicolor mode

Description

In multicolor mode, you're allowed to have the background color plus three foreground colors (instead of one). This routine sets up the additional colors.

Prototype

In a loop, read the three color values from MTCOLS and store them beginning at location 53281 (BGCOL0).

Explanation

To set multicolor-mode colors, choose three color values for the background color registers (53281–53283) and define them in MTCOLS at the end of the program. The program below is just a program fragment. For a complete example routine, see **MTCMOD**.

Routine

```
C000                BGCOL0    =      53281        ; text background color register 0
C000  A2  02        MTCCOL    LDX    #2           ; as an index
C002  BD  10  C0    COLOOP    LDA    MTCOLS,X     ; get each color value
C005  9D  21  D0              STA    BGCOL0,X     ; assign it to a register
C008  CA                      DEX                 ; for next register
C009  10  F7                  BPL    COLOOP       ; do all three
C00B  60                      RTS
                                                  ;
C00C  08  09  0A    COLORS    .BYTE  8,9,10,14    ; color orange, brown, light red, light blue
C010  0F  05  03    MTCOLS    .BYTE  15,5,3       ; color light gray, green, cyan
```

See also XBCCOL, XBCMOD, MTCMOD.

Name

Turn multicolor mode on or off

Description

Setting bit 4 in location 53270 (SCROLX) enables multicolor mode, which applies in both text or bitmap mode. The program below uses **MTCMOD** and **MTCCOL** to select multicolor text mode and set character colors for this mode.

Prototype

1. Load the contents of the horizontal fine-scrolling/control register at 53270 (SCROLX) into the accumulator.
2. ORA with %00010000 to turn on bit 4 and store the result back into the register. (To turn off multicolor mode, AND the contents of SCROLX with %11101111.)

Explanation

It's true that bit 4 of SCROLX enables multicolor mode. But in text mode, each individual character must have a value greater than 7 in its color RAM nybble before the character actually displays in multicolor. When this occurs, the horizontal resolution of each character is cut in half. Instead of having eight separate pixels across that can be one of two colors, the character is represented horizontally by four groups of double-width pixels. And the color of each double-width pixel is taken from one of four locations, depending on its bit pattern:

00	Background color register 0 at 53281
01	Background color register 1 at 53282
10	Background color register 2 at 53283
11	Bits 0–2 of corresponding color RAM nybble (55296–56319)

To see this effect, run the example program below. This program prints the characters *A–Z* four times in multicolor mode, varying color RAM on each pass. Looking at the results should convince you that the built-in character set was not intended to be used with multicolor mode. To take advantage of this feature in text mode, you'll need to design your own four-color characters with a routine such as **CHRDEF**.

If you turn on bitmapping (see **BITMAP**) at the same time multicolor mode is active, again double-width pixels will have the effect of halving horizontal screen resolution. But in bitmap mode, the color sources for the double-width pixels differ from text mode. Color sources for the four possible bit patterns are as follows:

00 Background color register 0 at 53281
01 High nybble of corresponding color byte
10 Low nybble of corresponding color byte
11 Bits 0–3 of corresponding color RAM nybble (55296–56319)

Note: On the 128, location 216, or GRAPHM, is copied into SCROLX during the screen-setup portion of the IRQ interrupt routine. You can prevent this altogether by storing a 255 in GRAPHM. If you allow the IRQ routine to copy GRAPHM, select multicolor mode from this register by setting bit 7.

The program below uses the first approach. So, 128 users should include the instructions LDA #$FF:STA GRAPHM just prior to activating multicolor mode in $C005.

Routine

```
C000                BGCOL0   =    53281        ; text background color register 0
C000                SCROLX   =    53270        ; scroll/control register
C000                CHROUT   =    65490
C000                GETIN    =    65508
C000                COLOR    =    646          ; COLOR = 241 on the 128—text foreground
                                               ; color register
C000                GRAPHM   =    216          ; mode flag for 40-column screen (128 only)
                                               ;
                                               ; Display characters A–Z four times in
                                               ; multicolor mode. Change foreground
                                               ; text color each time. Exit on keypress.
C000  A9 93         CHRCLR   LDA  #147         ; clear the screen
C002  20 D2 FF               JSR  CHROUT
                                               ; LDA #$FF; disable screen-setup portion of
                                               ; IRQ routine (add for 128 only)
                                               ; STA GRAPHM; (128 only)
C005  20 38 C0               JSR  MTCMOD       ; turn on multicolor mode
C008  20 41 C0               JSR  MTCCOL       ; assign multicolor mode colors
C00B  A2 03                  LDX  #3           ; print A–Z four times
C00D  BD 4D C0      AZLOOP   LDA  COLORS,X     ; get each text foreground color
C010  8D 86 02               STA  COLOR        ; store in the register
C013  A9 41                  LDA  #65          ; begin with A
C015  20 D2 FF      PRTLOP   JSR  CHROUT       ; display characters A–Z
C018  18                     CLC               ; for next character code
C019  69 01                  ADC  #1
C01B  C9 5B                  CMP  #91          ; is it Z plus 1?
C01D  D0 F6                  BNE  PRTLOP       ; and continue
C01F  A9 0D                  LDA  #13          ; carriage return
C021  20 D2 FF               JSR  CHROUT       ; print it twice
C024  20 D2 FF               JSR  CHROUT
C027  CA                     DEX               ; for next A–Z printing
C028  10 E3                  BPL  AZLOOP
C02A  20 E4 FF      GETKEY   JSR  GETIN        ; wait for a keypress
C02D  F0 FB                  BEQ  GETKEY       ; if no keypress, then wait
C02F  AD 16 D0               LDA  SCROLX       ; turn off multicolor mode
C032  29 EF                  AND  #%11101111
C034  8D 16 D0               STA  SCROLX       ; reset register
C037  60                     RTS
                                               ;
                                               ; Turn on (or off) multicolor mode.
```

```
C038  AD 16  D0 MTCMOD  LDA   SCROLX       ; get current register value
C03B  09 10             ORA   #%00010000   ; turn on bit 4 (turn off with AND
                                           ; %11101111)
C03D  8D 16  D0         STA   SCROLX       ; and set the register
C040  60                RTS
                                           ;
                                           ; Assign colors to multicolor color registers
                                           ; 53281-53283
C041  A2 02     MTCCOL  LDX   #2           ; as an index
C043  BD 51  C0 COLOOP  LDA   MTCOLS,X     ; get each color value
C046  9D 21  D0         STA   BGCOL0,X     ; assign it to a register
C049  CA                DEX                ; for next register
C04A  10 F7             BPL   COLOOP       ; do all three
C04C  60                RTS
                                           ;
C04D  08 09  0A COLORS  .BYTE 8,9,10,14    ; colors—orange, brown, light red, light blue
C051  0F 05  03 MTCOLS  .BYTE 15,5,3       ; colors—light gray, green, cyan
```

See also XBCCOL, XBCMOD, MTCCOL.

Name

Multiply two numbers with successive adds

Description

One way to multiply two numbers is to add one number to itself over and over. This technique works best on single bytes. As the numbers get larger, the time used by the routine increases to the point where it becomes very slow.

Prototype

1. Before calling the routine, store in memory the numbers to be multiplied.
2. Zero out the two-byte total.
3. Load the two numbers into .A and .X.
4. If either number is zero, exit the routine.
5. Decrement .X and exit when it hits zero.
6. Add the accumulator to the first number.
7. If the carry flag is set, indicating that the low byte overflowed, increment the high byte.
8. Loop back to step 5.

Explanation

The framing routine just gets two keypresses and stores the ASCII values of the characters in B1 and B2. Press Q to quit.

Within **MULAD1**, the two bytes of TOTAL are zeroed out; then the numbers in B1 and B2 are multiplied. If either number equals zero, the routine ends (with zeros still in TOTAL), because zero times any number is zero. As .X counts down to zero, the accumulator is repeatedly added to the number in B2.

Note: This approach to multiplying works reasonably well when the two numbers are byte-sized (0–255). If you need to multiply larger numbers, repeated addition becomes very slow. For example, multiplying 20,000 by 20,000 would require 20,000 iterations. Even at machine language speeds, this would take some time. For multiplying larger numbers, see **MULSHF**.

Routine

```
C000                LINPRT   =     $BDCD     ; LINPRT = $8E32 on the 128—ROM
                                             ; routine to print a number
C000                GETIN    =     $FFE4
C000                CHROUT   =     $FFD2
                                             ;
C000  20  E4  FF    MAIN     JSR   GETIN     ; get a key
C003  F0  FB                 BEQ   MAIN      ; wait until there's one there
C005  C9  51                 CMP   #81       ; check for Q (quit)
C007  F0  3D                 BEQ   QUIT
C009  8D  6D  C0             STA   B1        ; store it in byte 1
```

```
C00C  20  E4  FF  M2       JSR    GETIN       ; get another key
C00F  F0  FB               BEQ    M2
C011  C9  51               CMP    #81         ; check Q again
C013  F0  31               BEQ    QUIT
C015  8D  6E  C0           STA    B2          ; store in byte 2
C018  AE  6D  C0           LDX    B1
C01B  A9  00               LDA    #0
C01D  20  CD  BD           JSR    LINPRT      ; print number 1
C020  A9  2A               LDA    #42         ; the * character
C022  20  D2  FF           JSR    CHROUT      ; print it
C025  AE  6E  C0           LDX    B2          ; second number
C028  A9  00               LDA    #0
C02A  20  CD  BD           JSR    LINPRT      ; print it, also
C02D  A9  3D               LDA    #61         ; equal sign
C02F  20  D2  FF           JSR    CHROUT      ; print it
C032  20  47  C0           JSR    MULAD1      ; multiply the numbers
C035  AE  6F  C0           LDX    TOTAL       ; low byte
C038  AD  70  C0           LDA    TOTAL+1     ; high byte
C03B  20  CD  BD           JSR    LINPRT      ; print it
C03E  A9  0D               LDA    #13         ; <RETURN>
C040  20  D2  FF           JSR    CHROUT      ; print and
C043  4C  00  C0           JMP    MAIN        ; go back
C046  60          QUIT     RTS
                                              ;
C047  A9  [redacted]      MULAD1   LDA    #0          ; clear out
C049  8D  6F  C0           STA    TOTAL       ; low byte of total
C04C  8D  70  C0           STA    TOTAL+1     ; and high byte
                                              ;
C04F  AE  6D  C0           LDX    B1          ; the counter for repeated adds
C052  F0  18               BEQ    MULEND      ; if zero, no addition
C054  18                   CLC
C055  AD  6E  C0           LDA    B2          ; second number (which will be added)
C058  F0  12               BEQ    MULEND      ; if zero, no operation is necessary
C05A  CA          MULLOP   DEX                ; decrement .X first, in case it's a 1
C05B  F0  0C               BEQ    MULSTR      ; if zero, store the result in total (low byte)
C05D  18                   CLC                ; get ready
C05E  6D  6E  C0           ADC    B2          ; and add .A to B2
C061  90  F7               BCC    MULLOP      ; if carry is clear, no overflow to the high
                                              ; byte
C063  EE  70  C0           INC    TOTAL+1     ; else add one to high byte
C066  4C  5A  C0           JMP    MULLOP      ; and go back
                                              ;
C069  8D  6F  C0  MULSTR   STA    TOTAL       ; store the low byte (high byte is OK)
C06C  60          MULEND   RTS                ; and leave the routine
                                              ;
C06D  00          B1       .BYTE  0
C06E  00          B2       .BYTE  0
C06F  00  00      TOTAL    .BYTE  0,0
```

See also MULAD2, MULFP, MULSHF.

Name

Multiply two numbers with repeated addition (optimized version)

Description

This routine is basically the same as **MULAD1**, but the smaller number is placed in the X register to speed up the DEX loop. The larger number is repeatedly added to itself, and the result is stored in memory.

Prototype

1. Start by storing the two numbers in memory.
2. Store zeros in the two bytes of TOTAL.
3. Initialize .Y to zero on the assumption that the first number is larger.
4. Load .X with B2 and compare it with B1.
5. If B2 is smaller, branch forward to step 7.
6. Otherwise, load .X with B1 and change .Y to 1.
7. Load .A from B1, indexed by .Y.
8. Decrement .X and branch out of the routine when it's zero.
9. Add the accumulator to B1,Y.
10. Increment the high byte of TOTAL whenever the carry flag is set.

Explanation

The routine **MULAD1** is simpler than this one, but **MULAD2** is faster in certain situations. Take the example of 252 × 3. The simpler version of MULAD might calculate it by adding 252 to itself 3 times. Or it might add 3 to itself 252 times. Obviously, 3 additions execute faster than 252.

MULAD2 checks the size of the two numbers and puts the smaller into .X for the main loop. The Y register is used as an offset into the table of numbers; its value is either zero or one.

Note: As with **MULAD1**, the larger the values, the longer the time needed to repeatedly add the two numbers. For values larger than 255, **MULSHF** is preferable.

Routine

```
C000                LINPRT   =     $BDCD       ; LINPRT = $8E32 on the 128—ROM
                                               ; routine to print a number
C000                GETIN    =     $FFE4
C000                CHROUT   =     $FFD2
                                               ;
C000  20  E4  FF    MAIN     JSR   GETIN       ; get a key
C003  F0  FB                 BEQ   MAIN        ; wait until there's one there
C005  C9  51                 CMP   #81         ; check for Q (quit)
```

```
C007  F0 3D               BEQ   QUIT
C009  8D 77 C0            STA   B1         ; store it in byte 1
C00C  20 E4 FF   M2       JSR   GETIN      ; get another key
C00F  F0 FB               BEQ   M2
C011  C9 51               CMP   #81        ; check Q again
C013  F0 31               BEQ   QUIT
C015  8D 78 C0            STA   B2         ; store in byte 2
C018  AE 77 C0            LDX   B1
C01B  A9 00               LDA   #0
C01D  20 CD BD            JSR   LINPRT     ; print number 1
C020  A9 2A               LDA   #42        ; the * character
C022  20 D2 FF            JSR   CHROUT     ; print it
C025  AE 78 C0            LDX   B2         ; second number
C028  A9 00               LDA   #0
C02A  20 CD BD            JSR   LINPRT     ; print it also
C02D  A9 3D               LDA   #61        ; equal sign
C02F  20 D2 FF            JSR   CHROUT     ; print it
C032  20 47 C0            JSR   MULAD2     ; multiply the numbers
C035  AE 79 C0            LDX   TOTAL      ; low byte
C038  AD 7A C0            LDA   TOTAL+1    ; high byte
C03B  20 CD BD            JSR   LINPRT     ; print it
C03E  A9 0D               LDA   #13        ; <RETURN>
C040  20 D2 FF            JSR   CHROUT     ; print and
C043  4C 00 C0            JMP   MAIN       ; go back
C046  60         QUIT     RTS
                                           ;
C047  A9 00      MULAD2   LDA   #0         ; clear out
C049  8D 79 C0            STA   TOTAL      ; low byte of TOTAL
C04C  8D 7A C0            STA   TOTAL+1    ; and high byte
                                           ;
C04F  A8                  TAY              ; zero into .Y also
C050  AE 78 C0            LDX   [illegible] ; check B2
C053  F0 21               BEQ   MULEND     ; if zero, quit
C055  EC 77 C0            CPX   B1         ; is it smaller than B1?
C058  90 07               BCC   GOAHEAD    ; yes, continue
C05A  AE 77 C0            LDX   B1         ; else, B1 is the counter
C05D  F0 17               BEQ   MULEND     ; if zero, no need to multiply
C05F  A0 01               LDY   #1         ; and .Y is one instead of zero
                                           ;
C061  B9 77 C0   GOAHEAD  LDA   B1,Y       ; get the bigger number for adding
C064  CA         LOOP     DEX              ; check for possibility .X is one
C065  F0 0C               BEQ   MULSTR     ; if zero, store the low byte
C067  18                  CLC              ; else
C068  79 77 C0            ADC   B1,Y       ; add .A to B1
C06B  90 F7               BCC   LOOP       ; if carry clear, OK
C06D  EE 7A C0            INC   TOTAL+1    ; or add to the high byte
C070  4C 64 C0            JMP   LOOP
C073  8D 79 C0   MULSTR   STA   TOTAL      ; store the low byte
C076  60         MULEND   RTS              ; and return
                                           ;
C077  00         B1       .BYTE 0
C078  00         B2       .BYTE 0
C079  00 00      TOTAL    .BYTE 0,0
```

See also MULAD1, MULFP, MULSHF.

Name

Multiply two floating-point numbers

Description

The example program multiplies two numbers in floating-point format. It relies heavily on ROM routines.

Prototype

1. Put one number in floating-point accumulator 1 (FAC1).
2. Put the other in FAC2.
3. Call the FMULT routine. The result is in FAC1.

Explanation

The framing program sets up the numbers 10,000 and 11,111 in the two floating-point accumulators and multiplies them. The answer is printed to the screen.

The various ROM routines include GIVAYF (translate an integer from .A and .Y to a floating-point number in FAC1), MOVEF (move the contents of FAC1 to FAC2), FMULT (multiply FAC1 by FAC2), and FOUT (convert FAC1 to ASCII numbers).

Most of the time, you can write programs using integer values only. But if you find the need for floating-point numbers, it's generally easier to use the built-in ROM routines instead of writing your own. For a complete list of ROM routines and documentation on how they work, see *Mapping the Commodore 64* and *Mapping the Commodore 128* (both from COMPUTE! Publications).

Routine

```
C000             ZP       =   $FB
C000             CHROUT   =   $FFD2
C000             FMULT    =   $BA30     ; FMULT = $8A08 on the 128—multiply
                                        ; FAC2 and FAC1, result in FAC1
C000             MOVEF    =   $BC0F     ; MOVEF = $8C3B on the 128—move FAC1
                                        ; to FAC2
C000             GIVAYF   =   $B391     ; GIVAYF = $AF03 on the 128—convert
                                        ; integer to floating point
C000             FOUT     =   $BDDD     ; FOUT = $8E42 on the 128—convert FAC1
                                        ; to ASCII string
                                        ;
                                        ; Convert the numbers 10000 and 11111 to
                                        ; floating point and multiply
C000  A9 27                LDA  #>10000 ; high byte of 10000
C002  A0 10                LDY  #<10000 ; low byte
C004  20 91 B3             JSR  GIVAYF  ; convert it; now it's in FAC1
C007  20 0F BC             JSR  MOVEF   ; move FAC1 to FAC2
C00A  A9 2B                LDA  #>11111 ; high byte of 11111
C00C  A0 67                LDY  #<11111 ; low byte
```

```
C00E  20  91  B3           JSR   GIVAYF      ; convert it
                                             ; FAC2 now holds 10000, and FAC1 holds
                                             ; 11111.
C011  20  29  C0           JSR   MULFP       ; multiply them, with the result in FAC1
C014  20  DD  BD           JSR   FOUT        ; convert to ASCII
C017  85  FB               STA   ZP          ; pointer
C019  84  FC               STY   ZP+1        ; to the string
C01B  A0  00               LDY   #0
C01D  B1  FB      PRTLOP   LDA   (ZP),Y
C01F  D0  01               BNE   PRNIT
C021  60                   RTS
C022  20  D2  FF  PRNIT    JSR   CHROUT
C025  C8                   INY
C026  D0  F5               BNE   PRTLOP
C028  60                   RTS
                                             ;
C029  20  30  BA  MULFP    JSR   FMULT       ; multiply FAC2 by FAC1
C02C  60                   RTS               ; the result is in FAC1
```

See also MULAD1, MULAD2, MULSHF.

Name

Multiply two unsigned integer values using bit shifts

Description

MULSHF is a little more complex—and more difficult to understand—than the routines that multiply with successive additions (**MULAD1** and **MULAD2**), but it's much faster if you have large numbers to multiply.

Prototype

1. Start with the two numbers to be multiplied in B1 and B2 (16 bits each).
2. Store zeros in the 32 bits of TOTAL.
3. Copy B2 to WORK, a temporary storage area.
4. Store the number of bits to shift in COUNTR.
5. Shift WORK to the left.
6. If the carry flag is clear, skip step 7.
7. If it's set, add B1 to TOTAL.
8. Decrement the counter. If not zero, multiply TOTAL by two with right shifts.
9. If it is zero, exit. Otherwise, branch back to step 5.

Explanation

An expanded diagram of multiplying two four-bit numbers may be helpful:

```
1          1110
B2         1011
S4        1110
S3       1110
S2      0000
S1     1110
TOTAL  10011010
```

Start with the TOTAL equal to zero. Shift B2 to the left, and a one appears in the carry flag. That means it's time to add B1 to the total, which becomes S1 (00001110). There's more, so shift the total to the left (00011100). Shift B2 left again. This time there's a zero, so skip the addition, but shift TOTAL left again to become subtotal 2—S2 (00111000). Shift B2 left again, and carry is set; so add 1110 (01000110) and shift it left (10001100). Finally, shift B2 the final time, and carry is set, so add one more time (10011010), but don't shift the total to the left because it's the last addition.

By the same logic, multiplying 16-bit numbers requires 16

shifts. B1 and B2 each have 16 bits, so the total needs 32 bits. Note in the example above that multiplying two 4-bit numbers yields an 8-bit result. In general, when you multiply two numbers of a given size, the largest possible result will need double the number of bits. (Multiplying two 8-bit numbers results in a number that may be as large as 16 bits.)

Routine

```
C000  A0 03        MULSHF  LDY   #3          ; four bytes
C002  A9 00                LDA   #0          ; zero out TOTAL
C004  99 5C C0     ZOUT    STA   TOTAL,Y     ; store it
C007  88                   DEY               ; count down
C008  10 FA                BPL   ZOUT        ; and loop back
C00A  AD 58 C0             LDA   B2          ; copy B2 to WORK
C00D  8D 5A C0             STA   WORK
C010  AD 59 C0             LDA   B2+1
C013  8D 5B C0             STA   WORK+1
                                             ;
C016  A9 10                LDA   #16         ; there are 16 shifts, so
C018  8D 55 C0             STA   COUNTR      ; set up a counter
                                             ;
C01B  0E 5A C0     MULLP   ASL   WORK        ; shift the low byte
C01E  2E 5B C0             ROL   WORK+1      ; into the high byte
C021  90 1D                BCC   BIGSHF      ; if the bit is off, skip the add
C023  18                   CLC               ; clear carry before add
C024  AD 56 C0             LDA   B1          ; low byte
C027  6D 5C C0             ADC   TOTAL       ; add to TOTAL (low)
C02A  8D 5C C0             STA   TOTAL       ; store it
C02D  AD 57 C0             LDA   B1+1        ; second byte of four
C030  6D 5D C0             ADC   TOTAL+1     ; add it
C033  8D 5D C0             STA   TOTAL+1     ; store it
C036  90 08                BCC   BIGSHF      ; if carry clear, branch forward
C038  EE 5E C0             INC   TOTAL+2     ; else add 1 to third byte
C03B  D0 03                BNE   BIGSHF      ; if not zero, skip the fourth
C03D  EE 5F C0             INC   TOTAL+3     ; else, get the fourth
                                             ;
C040  CE 55 C0     BIGSHF  DEC   COUNTR      ; count down
C043  D0 01                BNE   SHIFIT      ; shift it if there's more
C045  60                   RTS               ; else, quit
C046  0E 5C C0     SHIFIT  ASL   TOTAL       ; multiply by 2
C049  2E 5D C0             ROL   TOTAL+1     ; all.
C04C  2E 5E C0             ROL   TOTAL+2     ; four...
C04F  2E 5F C0             ROL   TOTAL+3     ; bytes
C052  4C 1B C0             JMP   MULLP       ; repeat it again
                                             ;
C055  00           COUNTR  .BYTE 0
C056  7D 00        B1      .BYTE 125,0       ; value of 125
C058  58 02        B2      .BYTE 88,2        ; value of 600
C05A  00 00        WORK    .BYTE 0,0
C05C  00 00 00     TOTAL   .BYTE 0,0,0,0
```

See also MULAD1, MULAD2, MULFP.

Name

Move a block of data upward in memory

Description

MVU64 moves a block of data in memory from a lower to a higher address on the 64, even if the two blocks overlap. This routine can be used to relocate other machine language routines, as in the program below, or to move text and numerical-data tables. Assuming your source and destination blocks don't overlap, you could also move memory down with this routine.

Prototype

1. Store the ending address for the source block (BLOCK1) in ZP and the ending address for the target block (MEMSIZ −1) in ZP+2.
2. Store the number of bytes to move down (NUMBER, as calculated by the assembler) in .X (low byte) and .Y (high byte).
3. Store the number of bytes to move, currently in .X and .Y, into a two-byte counter (COUNTR).
4. Using indirect addressing in UPLOOP, transfer bytes from the source memory block (at ZP) to the target memory block (at ZP+2).
5. Decrease both zero-page pointers by one with the subroutine SUBONE.
6. Decrement the bytes counter (COUNTR) continuing UPLOOP until all bytes from the source block have been moved. Then RTS.

Explanation

In the program below, **MVU64** moves a relocatable ML program (the 16-byte CYCLE) to the top of BASIC. To guarantee that CYCLE moves up in memory, assemble this program in the cassette buffer at 828.

In moving memory, **MVU64** works backwards in memory from the end of the source block, transferring a byte at a time. Each byte, loaded from the source block, is in turn stored in the next-lowest position in the target block, until the entire block has been transferred.

In this program, we're locating CYCLE at the top of BASIC memory, so we use the top-of-BASIC pointer, or MEMSIZ, to determine the end of the target block. Since MEMSIZ actually points to the byte beyond the highest free

byte in the BASIC text area (normally, 40960), we subtract one before storing it to ZP+2.

Once CYCLE is positioned at the top of BASIC, MEMSIZ is adjusted to protect the relocated program from BASIC. At the same time, its SYS address is printed. To satisfy yourself that CYCLE has properly relocated, look at the 16 bytes of memory beginning with the SYS address, or simply SYS to it.

If you want to use **MVU64** to move memory down, switch the source and target block addresses stored in zero page. In other words, store the ending address for the source block in ZP+2, and the ending address for the target block in ZP. For this approach to be successful, the two memory blocks must not overlap.

NOTE: Unlike some memory-move routines (see **SWAPIT**), **MVU64** lacks error checking. So it's up to you to make sure the relative positions of the two memory blocks fulfill the requirements of the routine.

There is a BASIC ROM routine at $A3BF (about 50 bytes in length) on the 64 which will move memory up. Much like **MVU64**, if the source and destination blocks don't overlap, it also can move memory down. To implement it, load $5F–$60 with the starting address of the source block, load $5A–$5B with the source block's ending address plus 1, and load $58–$59 with the destination block's ending address plus 1. Then JSR to $A3BF.

Routine

```
033C                  ZP       =    251
033C                  GETIN    =    65508
033C                  CHROUT   =    65490
033C                  LINPRT   =    48589       ; BASIC two-byte number output
033C                  EXTCOL   =    53280       ; border-color register
033C                  MEMSIZ   =    55          ; top-of-BASIC pointer
                                                ;
                                                ; Move a relocatable ML program to the top
                                                ; of BASIC memory
033C  20 60 03                 JSR  MUINIT      ; initialize zero-page pointers and get number
                                                ; of bytes to move
033F  20 7A 03                 JSR  MVU64       ; move the program up
0342  A0 00                    LDY  #0          ; as an index in PRTLOP
0344  B9 A8 03  SYSLOP         LDA  SYSMSG,Y    ; get a character from SYSMSG
0347  F0 06                    BEQ  EXITPR      ; if a zero byte, then exit PRTLOP
0349  20 D2 FF                 JSR  CHROUT      ; print the character
034C  C8                       INY              ; for next character
034D  D0 F5                    BNE  SYSLOP      ; branch always
034F  18        EXITPR         CLC              ; for addition
0350  A5 FD                    LDA  ZP+2        ; get the low byte of relocated ML program
0352  69 01                    ADC  #1          ; add one since decremented in SUBONE one
                                                ; time too many
0354  85 37                    STA  MEMSIZ      ; at the same time, protect the ML program
                                                ; from BASIC
```

```
0356  AA                          TAX                 ; for low byte of LINPRT
0357  A5  FE                      LDA   ZP+3          ; get the high byte of relocated program
0359  69  00                      ADC   #0            ; add the carry flag value
035B  85  38                      STA   MEMSIZ+1
035D  4C  CD  BD  NUMOUT          JMP   LINPRT        ; print the SYS address and RTS
                                                      ;
                                                      ; Initialize ZP pointers to end of BLOCK1 and
                                                      ; top of BASIC. Two bytes at
                                                      ; ZP point to source, and two at ZP+2 point
                                                      ; to target. Also, put number of
                                                      ; bytes to move in .X and .Y
0360  A9  D6          MUINIT      LDA   #<BLOCK1      ; low byte of BLOCK1 first
0362  85  FB                      STA   ZP
0364  A2  03                      LDX   #>BLOCK1      ; then high byte
0366  86  FC                      STX   ZP+1
                                                      ; Now store ending address of target block in
                                                      ; ZP+2, ZP+3.
                                                      ; Subtract one from top-of-BASIC pointer so
                                                      ; it points to available storage.
0368  38                          SEC                 ; for subtraction
0369  A5  37                      LDA   MEMSIZ
036B  E9  01                      SBC   #1            ; subtract one from low byte
036D  85  FD                      STA   ZP+2          ; and store result in zero page
036F  A5  38                      LDA   MEMSIZ+1      ; get the high byte for top-of-BASIC pointer
0371  E9  00                      SBC   #0            ; to subtract carry
0373  85  FE                      STA   ZP+3          ; and store the result
0375  A2  10                      LDX   #<NUMBER      ; put low byte of number of bytes to move up
                                                      ; in .X
0377  A0  00                      LDY   #>NUMBER      ; and high byte in .Y
0379  60                          RTS
                                                      ;
                                                      ; Move bytes up. Enter with the number of
                                                      ; bytes to move in .X (low) and
                                                      ; .Y (high). End of source block is in two
                                                      ; bytes at ZP, and target in ZP+2
                                                      ; First store number to COUNTR
037A  8E  A6  03  MVU64           STX   COUNTR        ; store number to COUNTR, low byte first
037D  8C  A7  03                  STY   COUNTR+1      ; high byte's in .Y
0380  A0  00                      LDY   #0            ; as an index in UPLOOP
0382  B1  FB          UPLOOP      LDA   (ZP),Y        ; get a byte from end of source block
0384  91  FD                      STA   (ZP+2),Y      ; store it at the end of target block (top of
                                                      ; BASIC)
0386  20  99  03                  JSR   SUBONE        ; decrease ZP pointers by one
0389  CE  A6  03                  DEC   COUNTR        ; decrement counter low byte
038C  D0  F4                      BNE   UPLOOP        ; if low byte hasn't turned over, continue
                                                      ; moving memory up
038E  CE  A7  03                  DEC   COUNTR+1      ; otherwise, decrement the high byte
0391  AD  A7  03                  LDA   COUNTR+1      ; check the high byte to see if we've
                                                      ; reached the last page
0394  C9  FF                      CMP   #255          ; on the last page, high byte goes 0–255
0396  D0  EA                      BNE   UPLOOP        ; if not last page, continue
0398  60                          RTS
                                                      ;
                                                      ; Decrement zero-page pointers by one.
0399  C6  FB          SUBONE      DEC   ZP            ; decrement low byte of source
039B  D0  02                      BNE   DECTAR        ; if it hasn't turned over, handle target
                                                      ; pointers
039D  C6  FC                      DEC   ZP+1          ; decrement high byte
039F  C6  FD          DECTAR      DEC   ZP+2          ; do the same for target pointers
03A1  D0  02                      BNE   SBEXIT        ; if hasn't turned over, exit SUBONE
03A3  C6  FE                      DEC   ZP+3          ; decrement high byte, if necessary
03A5  60              SBEXIT      RTS
```

```
03A6  00 00      COUNTR   .WORD 0                    ; two-byte counter for remaining # of bytes
                                                     ; to move down
03A8  54 4F 20   SYSMSG   .ASC  "TO RUN RELOCATED PROGRAM, SYS "
                                                     ; SYS message
03C6  00                  .BYTE 0                    ; terminator byte
                                                     ;
                                                     ; Relocatable program to cycle border color
                                                     ; on a keypress. Quit on RETURN.
03C7  20 E4 FF   CYCLE    JSR   GETIN                ; check for a keypress
03CA  F0 FB               BEQ   CYCLE                ; no keypress
03CC  C9 0D               CMP   #13                  ; quit on RETURN
03CE  F0 06               BEQ   BLOCK1
03D0  EE 20 D0            INC   EXTCOL               ; otherwise, cycle border color
03D3  38                  SEC                        ; to always cause a branch
03D4  B0 F1               BCS   CYCLE
03D6  60         BLOCK1   RTS                        ; last byte of cycle routine is BLOCK1
                                                     ;
03D7             NUMBER   =     BLOCK1 - CYCLE + 1
                                                     ; let assembler calculate number of bytes in
                                                     ; cycle
```

See also MOVEDN, MVU128, SWAPIT.

Name

Move a block of data upward in memory

Description

MVU128 is practically identical to the routine **MVU64** in form and in function. Both routines move a chunk of memory from a lower address to a higher address. And both can be used to move memory down, provided the two memory blocks—source and destination—don't overlap.

Prototype

This is a two-part routine. In the initialization routine MUINIT:

1. Store the ending address for the source block (BLOCK1) in ZP and the ending address for the target block (FRERAM) in ZP+2.
2. Store the number of bytes to move down (NUMBER, as calculated by the assembler) in .X (low byte) and .Y (high byte).

In **MVU128**:

1. Store the number of bytes to move, currently in .X and .Y, into a two-byte counter (COUNTR).
2. Using indirect addressing in UPLOOP, transfer bytes from the source memory block (at ZP) to the target memory block (at ZP+2).
3. You can move memory up from one bank to another by defining BNKSRC (source bank number) and BNKTAR (target bank number) at the end of the program. Replace the LDA (ZP),Y at UPLOOP with the three instructions that follow it in the listing (currently in the form of comments) and the STA (ZP+2),Y just below this with the next four instructions (also given as comments).
4. Decrease both zero-page pointers by one with the subroutine SUBONE.
5. Decrement the bytes counter (COUNTR), continuing UPLOOP until all bytes from the source block have been moved. Then RTS.

Explanation

The example program is much like the one that illustrates **MVU64**. In both cases, we're moving the relocatable ML routine, CYCLE, higher in memory. The only difference is that in this case we're moving it to the top of a protected RAM area,

which begins at $1300 (normally, just below BASIC), whereas with **MVU64**, CYCLE was moved to the top of BASIC RAM. Rather than storing the end of BASIC pointer (minus 1) in ZP +2, here we load ZP+2 with FRERAM (7167).

In both programs, the basic description of the two routines themselves is the same. **MVU64** has a more thorough explanation.

Since **MVU128** also uses zero-page addressing, the routine can be adapted to move memory from bank to bank. This requires the Kernal routines INDFET and INDSTA. INDFET performs an indirect load into the accumulator from the bank in .X, while INDSTA stores .A indirectly into the bank in .X. To implement these routines, replace the LDA (ZP),Y at $0C3D with the commented instructions that follow (UPLOOP LDA #ZP:LDX BNKSRC:JSR INDFET) and replace the STA (ZP+2),Y at $0C3F with LDX #ZP+2:STX 697:LDX BNKTAR:JSR INDSTA. Also include the bank numbers for the source (BNKSRC) and target block (BNKTAR), defined at the end of the program.

Note: Because this routine doesn't check to see whether the two memory blocks are positioned properly in memory, be sure the memory block in ZP is lower in memory than the block addressed by ZP+2.

Routine

```
0C00               ZP      =    251
0C00               GETIN   =    65508
0C00               INDFET  =    65396       ; Kernal routine to load indirectly from any
                                            ; bank
0C00               INDSTA  =    65399       ; Kernal routine to store indirectly to any
                                            ; bank
0C00               CHROUT  =    65490
0C00               EXTCOL  =    53280       ; border color register
0C00               LINPRT  =    36402
0C00               FRERAM  =    7167        ; top of a free memory area protected from
                                            ; BASIC
                                            ;
                                            ; Move a relocatable ML program up to the
                                            ; top of free RAM area at $1300.
0C00 20 20 0C              JSR  MUINIT      ; initialize zero page pointers and get number
                                            ; of bytes to move
0C03 20 35 0C              JSR  MVU128      ; move the program up
0C06 A0 00                 LDY  #0          ; as an index in PRTLOP
0C08 B9 63 0C  SYSLOP      LDA  SYSMSG,Y    ; get a character from SYSMSG
0C0B F0 06                 BEQ  EXITPR      ; if a zero byte, then exit PRTLOP
0C0D 20 D2 FF              JSR  CHROUT      ; print the character
0C10 C8                    INY              ; for next character
0C11 D0 F5                 BNE  SYSLOP      ; branch always
0C13 18        EXITPR      CLC              ; for addition
0C14 A5 FD                 LDA  ZP+2        ; get the low byte of relocated ML program
0C16 69 01                 ADC  #1          ; add 1 since decremented in SUBONE one
                                            ; time too many
0C18 AA                    TAX              ; for low byte of LINPRT
```

```
0C19  A5 FE            LDA  ZP+3         ; get the high byte of relocated program
0C1B  69 00            ADC  #0           ; add the carry flag value
0C1D  4C 32 8E NUMOUT  JMP  LINPRT       ; print the SYS address and RTS
                                         ;
                                         ; Initialize ZP pointers to end of BLOCK1 and
                                         ; FRERAM. Two bytes at
                                         ; ZP point to source, and two at ZP+2 point
                                         ; to target. Also, put number of
                                         ; bytes to move in .X and .Y
0C20  A9 91    MUINIT  LDA  #<BLOCK1     ; low byte of BLOCK1 first
0C22  85 FB            STA  ZP
0C24  A2 0C            LDX  #>BLOCK1     ; then high byte
0C26  86 FC            STX  ZP+1
                                         ; Now store ending address of target block
                                         ; in ZP+2, ZP+3.
0C28  A9 EF            LDA  #<FRERAM     ; get low byte of top of free RAM
0C2A  85 FD            STA  ZP+2         ; and store it
0C2C  A9 1B            LDA  #>FRERAM     ; get high byte of top of free RAM
0C2E  85 FE            STA  ZP+3         ; and store it
0C30  A2 10            LDX  #<NUMBER     ; put low byte of number of bytes to move
                                         ; up in .X
0C32  A0 00            LDY  #>NUMBER     ; and high byte in .Y
0C34  60               RTS
                                         ;
                                         ; Move bytes up. Enter with the number of
                                         ; bytes to move in .X (low) and
                                         ; .Y (high). End of source block is in two
                                         ; bytes at ZP, and target in ZP+2.
                                         ; First store number to COUNTR
0C35  8E 61 0C MVU128  STX  COUNTR       ; store number to COUNTR, low byte first
0C38  8C 62 0C         STY  COUNTR+1     ; high byte's in .Y
                                         ;
0C3B  A0 00            LDY  #0           ; as an index in UPLOOP
0C3D  B1 FB    UPLOOP  LDA  (ZP),Y       ; get a byte from end of source block
                                         ;
                                         ; Substitute the next three lines for the
                                         ; previous line
                                         ; to move memory from bank to bank.
                                         ; UPLOOP LDA #ZP; put zero page pointer
                                         ; to end of source in .A
                                         ; LDX BNKSRC; bank number for source
                                         ; JSR INDFET; load indirectly from bank .X
                                         ; at the end of source
                                         ;
0C3F  91 FD            STA  (ZP+2),Y     ; store it at the end of target block (top of
                                         ; BASIC)
                                         ;
                                         ; Again, substitute the next four lines for
                                         ; the previous line
                                         ; to move from bank to bank.
                                         ; LDX #ZP+2; put zero-page pointer to
                                         ; target address in 697
                                         ; STX 697
                                         ; LDX BNKTAR; bank number for target
                                         ; JSR INDSTA; store indirectly from bank
                                         ; .X at end of target
                                         ;
0C41  20 54 0C         JSR  SUBONE       ; decrease ZP pointers by one
0C44  CE 61 0C         DEC  COUNTR       ; decrement counter low byte
0C47  D0 F4            BNE  UPLOOP       ; if low byte hasn't turned over, continue
                                         ; moving memory up
0C49  CE 62 0C         DEC  COUNTR+1     ; otherwise, decrement the high byte
0C4C  AD 62 0C         LDA  COUNTR+1     ; check the high byte to see whether we've
```

```
                                          ; reached the last page
0C4F  C9 FF            CMP   #255         ; on the last page, high byte goes from 0 to
                                          ; 255
0C51  D0 EA            BNE   UPLOOP       ; If not last page, continue
0C53  60               RTS
                                          ;
                                          ; Decrement zero-page pointers by one.
0C54  C6 FB     SUBONE DEC   ZP           ; decrement low byte of source
0C56  D0 02            BNE   DECTAR       ; if it hasn't turned over, handle target
                                          ; pointers
0C58  C6 FC            DEC   ZP+1         ; decrement high byte
0C5A  C6 FD     DECTAR DEC   ZP+2         ; do the same for target pointers
0C5C  D0 02            BNE   SBEXIT       ; if hasn't turned over, exit SUBONE
0C5E  C6 FE            DEC   ZP+3         ; decrement high byte, if necessary
0C60  60        SBEXIT RTS
                                          ;
0C61  00 00     COUNTR .WORD0             ; two-byte counter for remaining number of
                                          ; bytes to move down
0C63  54 4F 20  SYSMSG .ASC  "TO RUN RELOCATED PROGRAM, SYS "
                                          ; SYS message
0C81  00               .BYTE 0            ; terminator byte
                                          ; BNKSRC .BYTE 0; the bank number where
                                          ; source is
                                          ; BNKTAR BYTE 0, the bank number where
                                          ; target is
                                          ;
                                          ; Relocatable program to cycle border color
                                          ; on a keypress. Quit on RETURN.
0C82  20 E4 FF  CYCLE  JSR   GETIN        ; check for a keypress
0C85  F0 FB            BEQ   CYCLE        ; no keypress
0C87  C9 0D            CMP   #13          ; quit on RETURN
0C89  F0 06            BEQ   BLOCK1
0C8B  EE 20 D0         INC   EXTCOL       ; otherwise, cycle border color
0C8E  38               SEC                ; to always cause a branch
0C8F  B0 F1            BCS   CYCLE
0C91  60        BLOCK1 RTS                ; last byte of cycle routine is BLOCK1
                                          ;
0C92            NUMBER =     BLOCK1 - CYCLE + 1
                                          ; let assembler calculate number of bytes in
                                          ; cycle
```

See also MOVEDN, MVU64, SWAPIT.

Name

Set up an NMI interrupt routine

Description

NMIINT redirects the NMI interrupt vector to your own routine. This lets you wedge a custom routine into the normal NMI interrupt handler.

Prototype

Store the address of your custom NMI routine into the NMI interrupt vector and return to the calling program.

Explanation

The following program shows how to insert your own NMI interrupt routine (here, WEDGE). Once **NMIINT** has stored the address of your routine into the NMI interrupt vector at 792, anytime an NMI interrupt occurs—for instance, when you press RESTORE—your routine will execute before the normal interrupt handler is serviced.

In this case, within WEDGE, the cursor, border, and background colors for the screen are reset to the default values defined at the end of the program (in DCOLOR, DEXTCL, and DBGCOL). Currently, the background and border colors default to black while the cursor becomes light blue. If you'd prefer different colors, substitute the appropriate color values found in the table under **COLFIL**.

The 64 requires that certain registers—specifically, the A, X, and Y registers—be maintained while the NMI interrupt is being serviced. At the outset of WEDGE, then, these registers are saved on the stack. And at the end of the routine, they're restored.

The 128 also maintains these registers, along with the current bank configuration, while the NMI interrupt is serviced. But on the 128, these registers are actually saved prior to jumping through the NMI interrupt vector. Consequently, you don't have to worry about maintaining them yourself during the custom interrupt routine.

Routine

```
C000        NMIVEC   =   792      ; vector to nonmaskable interrupt routine
C000        NMINOR   =   65095    ; NMINOR = 64064 on the 128—normal
                                  ; NMI handler routine
C000        COLOR    =   646      ; COLOR = 241 on the 128—current text
                                  ; foreground color
C000        EXTCOL   =   53280    ; border color register
C000        BGCOL0   =   53281    ; screen background color register
                                  ;
```

```
                                                        ; Set default screen, border, cursor color on
                                                        ; RESTORE key.
C000  A9 0B       NMIINT   LDA   #<WEDGE                ; redirect NMI vector to our routine, low
                                                        ; byte first
C002  8D 18 03             STA   NMIVEC
C005  A9 C0                LDA   #>WEDGE                ; then high byte
C007  8D 19 03             STA   NMIVEC+1
C00A  60                   RTS                          ; we're done
                                                        ;
                                                        ; Restore default colors.
C00B  48          WEDGE    PHA                          ; push .A, .X, and .Y onto the stack (not
                                                        ; necessary on the 128)
C00C  8A                   TXA                          ; push .X
C00D  48                   PHA
C00E  98                   TYA                          ; push .Y
C00F  48                   PHA
                                                        ; Now restore colors.
C010  AD 2A C0             LDA   DCOLOR                 ; cursor first
C013  8D 86 02    TXTCOL   STA   COLOR
C016  AD 2B C0             LDA   DEXTCL                 ; then border color
C019  8D 20 D0    BORCOL   STA   EXTCOL
C01C  AD 2C C0             LDA   DBGCOL                 ; and lastly, screen color
C01F  8D 21 D0    BCKCOL   STA   BGCOL0
C022  68                   PLA                          ; restore the registers .Y, .X, and .A (not
                                                        ; necessary on the 128)
C023  A8                   TAY                          ; .Y first
C024  68                   PLA                          ; then .X
C025  AA                   TAX
C026  68                   PLA                          ; and finally .A
C027  4C 47 FE             JMP   NMINOR                 ; go to normal NMI handler
                                                        ;
C02A  0E          DCOLOR   .BYTE 14                     ; default cursor color of light blue
C02B  00          DEXTCL   .BYTE 0                      ; default border color of black
C02C  00          DBGCOL   .BYTE 0                      ; default screen color of black
```

See also IRQINT, RAS64, RAS128.

Name

Create a table of standard frequencies (eight octaves of 12 notes each)

Description

NOTETB generates a full table of two-byte frequencies representing the range of notes played by the SID chip. Once this table has been created, you can play musical tunes using notes from the table.

Prototype

1. Set up a frequency table (OCT7TB) containing the 12 standard notes in the highest octave (octave 7) and set aside 168 bytes below this for octaves 0–6 (FREQTB).
2. Position ZP at the beginning of OCT7TB, and ZP+2 at the start of what will be the sixth octave in FREQTB (24 bytes below OCT7TB).
3. Divide each two-byte note in OCT7TB by 2 and store the result in FREQTB as the corresponding note in the next lower octave.
4. Repeat Step 3, beginning with notes from the next lower octave each time, until FREQTB is complete.
5. Return from the routine.

Explanation

Each time you drop down an octave, the frequency for each note within that octave is half the value of the corresponding note in the octave above it. **NOTETB** uses this fact to generate the standard note table (FREQTB). Starting with notes from the highest octave, or octave 7, two-byte frequencies for each note in the octave below are calculated based on the preceding octave. This continues until the entire table—eight octaves of 12 notes each—is constructed.

When **NOTETB** is added to your music-playing routines, you can index frequencies from the table it generates by note number without having to type in all the frequencies yourself.

For instance, if you look at the program for **MELODY**, you'll see it uses a frequency table containing 15 notes (also labeled FREQTB). Frequencies within this table include all the notes from G-4 through A-5. In order to reference the frequencies in this table, a second table of note numbers (NOTES) is required.

In this case, 15 frequency values is not many to type yourself. But if you were playing more than one song or music

which had a wider range of notes, you'd be better off allowing **NOTETB** to build the frequency table for you.

The frequencies in the note table created by **NOTETB** are the same as those in the note table provided in the 64 and 128 *Programmer's Reference Guides*. Both tables contain 96 notes. As a result, you can use the tables in these reference guides to choose the appropriate note numbers for your music.

The only difference in the tables in the reference guides and the one created by **NOTETB** (FREQTB) is in the note-numbering system used to index the various frequencies. In FREQTB, the note numbers run continuously from 0–95. The note numbers in the reference guide tables, on the other hand, jump by 5 after each octave. Consequently, the numbers range from 0–123. To convert a note number from the reference guide tables to the number indexing the equivalent note in FREQTB, use the following formula:

NN = PRGNN − OCTAVE * 5

In this formula, PRGNN represents the note number taken from the table in the reference guide; OCTAVE, the octave number for the note (0–7); and NN, the number for the same note in FREQTB.

For example, middle C (C-4) in the reference guide tables is note number 64. To index this same note in FREQTB, use the number 64 − 4 * 5, or 44.

Routine

```
C000                ZP        =      251
                                           ;
                                           ; Create FREQTB by dividing each note in
                                           ; next higher octave by 2
C000  A9 E8   NOTETB  LDA  #<OCT7TB        ; position ZP at beginning of seventh
                                           ; octave (OCT7TB)
C002  85 FB           STA  ZP
C004  A9 C0           LDA  #>OCT7TB
C006  85 FC           STA  ZP+1
C008  A9 D0           LDA  #<OCT7TB-24     ; position ZP+2 at beginning of sixth
                                           ; octave
C00A  85 FD           STA  ZP+2
C00C  A9 C0           LDA  #>OCT7TB-24
C00E  85 FE           STA  ZP+3
C010  A2 07           LDX  #7              ; index for the octaves 0-6
C012  A0 17   OCTLOP  LDY  #23             ; position pointer on high byte of highest
                                           ; note in octave
C014  B1 FB   INLOOP  LDA  (ZP),Y          ; get the high byte of each note in octave
C016  4A              LSR                  ; divide it by 2
C017  91 FD           STA  (ZP+2),Y        ; store as the high byte of the note in the
                                           ; next lower octave
C019  88              DEY                  ; decrement pointer so it addresses the low
                                           ; byte of the note
```

```
C01A  B1  FB              LDA   (ZP),Y      ; get the low byte of each note in the
                                            ; octave
C01C  6A                  ROR               ; divide it by 2, handling carry from LSR
C01D  91  FD              STA   (ZP+2),Y    ; store as the low byte of the note in the
                                            ; next lower octave
C01F  88                  DEY               ; so pointer addresses high byte on the next
                                            ; pass
C020  10  F2              BPL   INLOOP      ; do until all 12 two-byte notes are handled
                                            ; Now subtract 24 so ZP and ZP+2 point to
                                            ; next-lower octaves.
C022  38                  SEC               ; subtract 24 from ZP
C023  A5  FB              LDA   ZP          ; low byte first
C025  E9  18              SBC   #24
C027  85  FB              STA   ZP
C029  A5  FC              LDA   ZP+1        ; then high byte
C02B  E9  00              SBC   #0
C02D  85  FC              STA   ZP+1
C02F  38                  SEC               ; now subtract 24 from ZP+2
C030  A5  FD              LDA   ZP+2        ; low byte first
C032  E9  18              SBC   #24
C034  85  FD              STA   ZP+2
C036  A5  FE              LDA   ZP+3        ; then high byte
C038  E9  00              SBC   #0
C03A  85  FE              STA   ZP+3
C03C  CA                  DEX               ; for next lower octave
C03D  D0  D3              BNE   OCTLOP      ; seven octaves complete frequency table
C03F  60                  RTS
                                            ;
C040             FREQTB   =     *           ; reserve room for lower seven octaves
C0E8                      *=    *+168       ; seven octaves of 12 two-byte notes
                                            ; OCT7TB is table of standard two-byte
                                            ; frequencies from the seventh octave.
C0E8  1E  86  18 OCT7TB   .WORD 34334,36376,38539,40830,43258,45830
C0F4  AC  BD  F3          .WORD 48556,51443,54502,57743,61176,64814
```

See also BEEPER, BELLRG, EXPLOD, INTMUS, MELODY, SIDCLR, SIDVOL, SIRENS.

Name

Print two-byte integer values

Description

NUMOUT prints a two-byte integer value in the range 0–65535 to the screen (or to the current output device). This general integer-printing routine is good for printing scores in games. It can also be useful for debugging programs. Suppose you want to know the effect your program is having on a two-byte address while the program is running. **NUMOUT** makes monitoring these locations a snap.

Prototype

1. Enter with .X containing the low byte and .A, the high byte of the two-byte integer value to be printed.
2. JMP to LINPRT to print the number and return.

Explanation

Relying on the BASIC ROM routine LINPRT keeps **NUMOUT** short and simple. If you're working on a 128, be sure to change the address of LINPRT to 36402.

Warning: If you use **NUMOUT** in a loop, index the loop by .Y rather than by .X, since its setup necessarily changes the contents of the X register.

Routine

```
C000              LINPRT  =     48589       ; LINPRT = 36402 on the 128
                                            ;
C000  AE 0C C0            LDX   INTGER      ; low byte of integer 85
C003  AD 0D C0            LDA   INTGER+1    ; high byte of 85
C006  4C 09 C0            JMP   NUMOUT      ; print the number and RTS
                                            ;
                                            ; Print the two-byte integer in .X (low byte)
                                            ; and .A (high byte).
C009  4C CD BD    NUMOUT  JMP   LINPRT      ; print the number and RTS
                                            ;
C00C  55 00       INTGER  .WORD 85          ; integer 85
```

See also BYTASC, CNUMOT, FACPRD, FACPRT.

Name

Open a sequential/program file

Description

Anytime you want to read or write data to the disk in the form of either a sequential or a program file, this is the first routine you'll need. **OPENFL** opens a designated channel to the disk for data transfer.

Prototype

1. On the 128, set the bank to 15 in the program which calls **OPENFL** (see **READBF** or **WRITBF**).
2. OPEN 1,8,2 with a sequential or program filename (SETLFS, SETNAM, OPEN). Then return to the calling program.
3. On the 128, prior to SETNAM, load the accumulator with the bank for the opened file and load the X register with the bank containing the program filename. Then JSR to SETBNK.

Explanation

In the example routine as it's given, we've opened the *sequential* file SEQUENTIAL for reading (,S,R). To open a *program* file for reading, add the suffix ,P,R to the filename. If the file that you open is to be written to, add the suffix ,S,W or ,P,W to the filename, depending on whether it's a sequential file or program file.

The logical file number assigned to the open channel below is 1. Any number from 1 through 255 will suffice, but it's best to use numbers less than 128. File numbers above 127 may cause line feed characters to be sent with each carriage return when performing a write operation.

For data transfers, any secondary address in the range 2–14 can be used. The device number value depends on how your drive is configured, but usually it's device 8 unless you have more than one drive.

On the 128, the program calling **OPENFL** must set the computer to bank 15 since Kernal routines are being used by this routine. Also be sure to set the bank number where the file is opened with BNKNUM and indicate to the routine the bank containing the filename by defining BNKFNM.

Note: Disk error checking can be incorporated into this routine, if needed. At the outset, OPEN the error channel.

Add **DERRCK** to the end of the program and JSR to it just after the JSR OPEN instruction.

Warning: Using **OPENFL** just opens a file, either sequential or program, for a read or write operation—no data is actually transferred. Complete example programs that read or write data to disk are offered elsewhere (see **READBF** to read a file, **WRITBF** to write one).

Routine

```
C000                SETLFS  =  65466
C000                SETNAM  =  65469
C000                OPEN    =  65472
C000                SETBNK  =  65384       ; Kernal bank number for OPEN and
                                           ; filename (128 only)
C000                MMUREG  =  65280       ; MMU configuration register (128 only)
                                           ;
                                           ; OPENFL opens a sequential or program file
                                           ; for reading or writing.
C000                OPENFL  =  *
                                           ; Set the 128 to bank 15 in the main
                                           ; program (see READBF or WRITBF).
                                           ;
                                           ; Open channel 15 here if you include error
                                           ; checking (DERRCK).
                                           ;
                    LDA     #1             ; logical file 1
C002  A2 08         LDX     #8             ; disk drive device number
C004  A0 02         LDY     #2             ; secondary address (2-14 are OK)
C006  20 BA FF      JSR     SETLFS         ; set file parameters
                                           ;
                                           ; Include the following three instructions
                                           ; on the 128
                                           ; LDA BNKNUM; bank number for data
                                           ; LDX BNKFNM; bank containing the
                                           ; ASCII filename
                                           ; JSR SETBNK
                                           ;
C009  A9 09         LDA     #FNLENG        ; length of filename
C00B  A2 16         LDX     #<FILENM       ; address of filename
C00D  A0 C0         LDY     #>FILENM
C00F  20 BD FF      JSR     SETNAM         ; set up filename
                                           ;
C012  20 C0 FF      JSR     OPEN           ; open the file for data transfer
                                           ;
                                           ; Insert JSR DERRCK here for disk error
                                           ; checking.
                                           ;
C015  60            RTS                    ; return to main program
                                           ;
                                           ; JSR DERRCK; Insert if including error
                                           ; checking.
                                           ;
```

```
C016  30  3A  53  FILENM   .ASC   "0:SEQUENTIAL,S,R"
                                                 ; sequential filename to open for a read
                                                 ; ,S,R is optional with sequential file reads.
                                                 ; Change to "0:PROGRAM,P,R" to open a
                                                 ; program file for reading.
C01F              FNLENG   =      *-FILENM       ; length of filename
                                                 ;
                                                 ; Include the next two variables on the 128
                                                 ; BNKNUM .BYTE 0; bank number where
                                                 ; data is found
                                                 ; BNKFNM .BYTE 0; bank number where
                                                 ; ASCII filename is located
```

See also READBF, READFL.

Name

Open a printer channel

Description

OPENPR opens a channel to the printer for subsequent output.

Prototype

1. OPEN the printer channel with the parameters 4,4,0 (SETLFS and OPEN).
2. Direct output to channel 4—load .X with the printer file number and JMP to CHKOUT.

Explanation

In the example program, the printer is opened as channel 4.

For an entire printer program, see **PRTOUT** for printing individual characters or **PRTSTR** for printing strings.

Note: For most printers, the logical file number for the output can be any integer in the range 0–255, while the device number is usually 4 (all Commodore printers are normally device 4). Some printers can also use 5 as a device number. The Commodore 1520 printer/plotter is device 6.

For Commodore printers, the secondary address sends information about the character set. A value of 0 causes Commodore printers to print in uppercase and graphics. A value of 7, on the other hand, causes them to print in uppercase and lowercase. Some printers require a value of 255 (for no secondary address) here. It is best to consult your printer manual and interface manual to determine the exact significance this parameter will have with your printer.

Routine

```
C000                SETLFS  =    65466
C000                OPEN    =    65472
C000                CHKOUT  =    65481
                                           ;
                                           ; Open a file to the printer as 4,4,0.
C000  A9 04         OPENPR  LDA  #4        ; logical file 4
C002  A2 04                 LDX  #4        ; device number for printer (change if
                                           ; printer uses another number)
C004  A0 00                 LDY  #0        ; secondary address
                                           ; A value of 0 here causes Commodore
                                           ; printers to print in uppercase/graphics.
                                           ; A value of 7 here causes Commodore
                                           ; printers to print in uppercase/lowercase.
                                           ; A value of 255 is required by some
                                           ; printers (meaning no secondary address).
                                           ;
```

```
C006  20  BA  FF        JSR   SETLFS     ; set values
C009  20  C0  FF        JSR   OPEN       ; open a file to printer
C00C  A2  04            LDX   #4
C00E  4C  C9  FF        JMP   CHKOUT     ; direct output to file 4 and RTS
```

See also CLOSFL, PRTOUT, PRTSTR.

Name

Fill an irregular hi-res enclosed outline with a solid color

Description

If you've drawn a series of lines or shapes on the hi-res screen, you can call this routine and fill in an enclosed shape with a solid color.

Prototype

1. Enter with a hi-res location specified in STARTX and STARTY.
2. Convert STARTX/STARTY to a memory location on the hi-res screen and a bitmask. Push the three bytes on the pseudostack.
3. Begin the fill: Pull a bitmask and memory location from the pseudostack. If the stack is empty, exit the routine.
4. Move to the left, looking for an edge of the enclosing shape.
5. Begin setting bits, moving to the right until a right-hand edge (or the edge of the screen) is discovered.
6. While the fill is proceeding, PEEK the bitmap locations above and below. Look first for an open (zero) bit.
7. When a zero is found, push that location and the bitmask on the pseudotack and set the FINDUP or FINDDN flag to search for ones.
8. If searching for a one, flip the FIND flag again (but don't save the address). Continue flipping the flag as you check the bits above and below.
9. When the main line is filled, go back to step 3.

Explanation

The routine, as it's written, uses no Kernal or ROM routines, so it will work on both 64s and 128s without modification. A note of interest to 128 owners: In a test of this machine language fill routine against the 128's BASIC 7.0 PAINT command, the BASIC command took an average of 70 seconds to fill most of the screen, while the routine below took only 10 seconds.

Drawing a straight line from left to right isn't difficult. The heart of the **PAINT** routine moves to the left until it finds an edge. Then it turns on pixels until it finds a right-hand edge of the outline being filled.

Simultaneously with the fill, the routine checks the pixels above and below, using two zero-page locations (ZU and ZD) that move in step with the fill. Consider just the pixel above.

We begin by looking for a zero. If ZU (plus the bitmask) points to a one (a pixel that's turned on), it's either the top edge of the figure or it's a previously filled line. We ignore all pixels that are on, at least at the beginning.

But if ZU points to a zero, then it will eventually have to be filled. So the address from ZU and the current bitmask (which rotates from right to left, from %10000000 to %00000001) are saved on the pseudostack. Now that we've found a zero, we can ignore any more zeros that pop up. The FINDUP flag is switched. Now we're searching for a one, because the fill routine will stop at an edge.

While we're looking for ones, we ignore zeros. When we find a pixel that is on, we have to flip the FINDUP flag again, to start looking for zeros. When a zero is discovered, save the address and mask, and flip the flag again. The process continues until the primary line runs up against an edge. At that point, we go back to the stack and start another fill. As long as there are more addresses, the paint routine is active.

The pseudostack is just an empty area of memory used to save the addresses. It follows the program, but you can change its location easily enough. For most shapes, a stack of two pages (512 bytes) should suffice.

To use this routine in your own programs, you'll need to change the variables at the end of the program. Store the first and last bytes of the bitmap area in BITMAP and BITMAX. The example assumes the hi-res screen runs from 8192 to 16191. Store a zero into FINDL if you're changing zeros to ones. Put a 255 there to clear bits from one to zero. And store a two-byte *x* location in STARTX (0–319) and a one-byte *y* location in STARTY (0–199) before you JSR to **PAINT**.

Routine

```
C000                SP          =     3
C000                ZU          =     $F9
C000                ZL          =     $FB
C000                ZD          =     $FD
                                                  ;
C000  A9  F8                    LDA   #<STACK     ; copy the stack address
C002  85  03                    STA   SP          ; to SP (stack pointer)
C004  A9  C1                    LDA   #>STACK
C006  85  04                    STA   SP + 1      ; high byte
C008  20  79  C0                JSR   CONVERT     ; change STARTX and STARTY to memory
                                                  ; location in the bitmap
                                                  ;
C00B  AD  F4  C1  BIGLOOP       LDA   FINDL       ; copy the FINDL mask to
C00E  8D  F5  C1                STA   FINDUP      ; the up mask
C011  8D  F6  C1                STA   FINDDN      ; and the down mask
                                                  ;
```

```
C014  20  E8  C0             JSR   PULLZL       ; pull ZL from the stack
C017  90  01                 BCC   NOTDONE      ; carry clear means there is more
C019  60                     RTS                ; if carry set, quit and RTS
C01A  20  16  C1  NOTDONE    JSR   LEFTZL       ; move left to find an edge
C01D  20  9A  C1             JSR   SETZUZD      ; set values for ZU and ZD (up and down)
                                                ;
C020              PAINT      =     *
C020  A0  00                 LDY   #0           ; set the index to zero
C022  B1  FB                 LDA   (ZL),Y       ; get the byte
C024  AA                     TAX                ; save in .X
C025  4D  F4  C1             EOR   FINDL        ; fix zeros or ones
C028  2D  EC  C1             AND   MASK         ; look at the bit
C02B  D0  49                 BNE   ENDPNT       ; we hit an edge, so quit
C02D  8A                     TXA                ; get the byte back
C02E  4D  EC  C1             EOR   MASK         ; and flip the bit
C031  91  FB                 STA   (ZL),Y       ; which is stored on the bitmap
                                                ;
                                                ; Now check the pixels above and below.
C033  AD  F5  C1  CKUP       LDA   FINDUP       ; get the search pattern
C036  AA                     TAX                ; put it into .X
C037  51  F9                 EOR   (ZU),Y       ; fix zeros or ones
C039  2D  EC  C1             AND   MASK         ; is it what we want?
C03C  D0  10                 BNE   CKDOWN       ; no, check the ZD pixel
                                                ; Found one, but is it off or on?
C03E  EC  F4  C1             CPX   FINDL        ; if it's not the same
C041  D0  03                 BNE   XORUP        ; move forward
C043  20  C4  C0             JSR   PUSHZU       ; else, push ZU on the pseudostack to
                                                ; handle later
C046  AD  F5  C1  XORUP      LDA   FINDUP       ; the FINDUP flag
C049  49  FF                 EOR   #$FF         ; gets flipped
C04B  8D  F5  C1             STA   FINDUP       ; and stored
                                                ;
C04E  AD  F6  C1  CKDOWN     LDA   FINDDN       ; check the down flag
C051  AA                     TAX                ; save it
C052  A0  00                 LDY   #0           ; .Y was altered by CKUP
C054  51  FD                 EOR   (ZD),Y
C056  2D  EC  C1             AND   MASK         ; same as above
C059  D0  10                 BNE   ZIBBL        ; it's OK
                                                ; Check the down bit. Off or on?
C05B  EC  F4  C1             CPX   FINDL        ; is it the same?
C05E  D0  03                 BNE   XORDN        ; no, skip it
C060  20  CA  C0             JSR   PUSHZD       ; yes, save the address
C063  AD  F6  C1  XORDN      LDA   FINDDN
C066  49  FF                 EOR   #$FF         ; switch FINDDN to its opposite
C068  8D  F6  C1             STA   FINDDN
                                                ;
C06B  20  6B  C1  ZIBBL      JSR   RIGHTZL      ; move ZL right a pixel
C06E  B0  06                 BCS   ENDPNT       ; CS means the line is done
C070  20  9A  C1             JSR   SETZUZD      ; we're OK, so do more
C073  4C  20  C0             JMP   PAINT        ; and go back
C076  4C  0B  C0  ENDPNT     JMP   BIGLOOP
                                                ;
C079              CONVERT    =     *
C079  AD  F1  C1             LDA   BITMAP+1     ; high byte of BITMAP address
C07C  85  FC                 STA   ZL+1         ; goes into high-byte of ZL pointer
C07E  AD  EF  C1             LDA   STARTY       ; y-position (0-199)
C081  A8                     TAY                ; stash into .Y
C082  29  07                 AND   #7           ; get the bottom three bits
C084  85  FB                 STA   ZL           ; store in low-byte of ZL
C086  98                     TYA                ; get it back
C087  4A                     LSR                ; shift right
C088  4A                     LSR                ; three times
C089  4A                     LSR
C08A  A8                     TAY                ; back into .Y
```

```
C08B  F0 10           BEQ   CONVX        ; If zero, skip ahead to do .X and
                                         ; x-coordinate
C08D  18        Y320  CLC
C08E  A9 40           LDA   #<320        ; else add 320
C090  65 FB           ADC   ZL           ; to ZL
C092  85 FB           STA   ZL           ; store it
C094  A9 01           LDA   #>320
C096  65 FC           ADC   ZL+1         ; high-byte, too
C098  85 FC           STA   ZL+1
C09A  88              DEY                ; count down
C09B  D0 F0           BNE   Y320         ; and branch back
                                         ;
C09D  AD ED C1 CONVX  LDA   STARTX       ; low byte of x-position
C0A0  AA              TAX                ; save in .X for a moment
C0A1  29 F8           AND   #%11111000   ; strip the three low bits
C0A3  18              CLC
C0A4  65 FB           ADC   ZL           ; add to ZL
C0A6  85 FB           STA   ZL           ; store
C0A8  AD EE C1        LDA   STARTX+1     ; get the high-byte
C0AB  65 FC           ADC   ZL+1         ; add to the high-byte
C0AD  85 FC           STA   ZL+1         ; save it
                                         ;
C0AF  A9 80           LDA   #%10000000   ; prepare mask
C0B1  8D EC C1        STA   MASK
C0B4  8A              TXA                ; get .X back
C0B5  29 07           AND   #%00000111   ; positions 0-7
C0B7  F0 07           BEQ   CONEXIT      ; if 0, skip it
C0B9  AA              TAX                ; else count down
C0BA  4E EC C1 CMASKL LSR   MASK         ; move MASK right
C0BD  CA              DEX                ; .X minus 1
C0BE  D0 FA           BNE   CMASKL       ; branch back
C0C0  20 C7 C0 CONEXIT JSR  PUSHZL       ; push the ZL and MASK bytes on the
                                         ; pseudostack
C0C3  60              RTS                ; and we're done
                                         ; As we leave, the location and the mask
                                         ; of the STARTX and STARTY points are
                                         ; on the stack.
C0C4  A2 01     PUSHZU LDX  #1
C0C6  2C              .BYTE $2C          ; BIT hides next instruction
C0C7  A2 03     PUSHZL LDX  #3
C0C9  2C              .BYTE $2C
C0CA  A2 05     PUSHZD LDX  #5
C0CC  A0 02           LDY   #2           ; three bytes (0-2)
C0CE  AD EC C1        LDA   MASK         ; get the mask
C0D1  91 03           STA   (SP),Y       ; store indirect to SP, which points to stack
C0D3  88              DEY                ; .Y is now 1
C0D4  B5 F9     PSHLP LDA   ZU,X         ; get a byte from ZU, ZL, or ZD
C0D6  91 03           STA   (SP),Y
C0D8  CA              DEX
C0D9  [illegible]     DEY
C0DA  10 F8           BPL   PSHLP        ; count 1 to 0 to minus
                                         ; Now adjust the stack pointer SP.
C0DC  18              CLC
C0DD  A9 03           LDA   #3           ; add 3
C0DF  65 03           ADC   SP           ; add to SP
C0E1  85 03           STA   SP           ; store it
C0E3  90 02           BCC   PSHOUT       ; carry clear, we're done
C0E5  E6 04           INC   SP+1
C0E7  60        PSHOUT RTS               ; Finished. Quit this routine.
                                         ;
C0E8  38        PULLZL SEC               ; first count SP down by 3
C0E9  A5 03           LDA   SP           ; low byte
C0EB  E9 03           SBC   #3           ; minus 3
C0ED  85 03           STA   SP           ; store it
```

```
C0EF  A5 04          LDA  SP+1        ; high byte
C0F1  E9 00          SBC  #0          ; minus zero (or one if carry clear)
C0F3  85 04          STA  SP+1        ; remember it
C0F5  C9 C1          CMP  #>STACK     ; check the high byte
C0F7  90 1B          BCC  ABORT       ; branch if the stack is empty
C0F9  D0 06          BNE  NOABORT     ; if not equal, keep going
C0FB  A5 03          LDA  SP          ; SP-high and STACK-high are equal, so
                                      ; check low byte
C0FD  C9 F8          CMP  #<STACK     ; against STACK
C0FF  90 13          BCC  ABORT       ; abort if STACK is higher (equal is OK)
C101  A0 02     NOABORT LDY [illegible]
C103  B1 03          LDA  (SP),Y      ; get the mask
C105  8D EC C1       STA  MASK        ; store it
C108  88             DEY              ; count down
C109  B1 03          LDA  (SP),Y
C10B  85 FC          STA  ZL+1        ; high byte of screen address
C10D  88             DEY
C10E  B1 03          LDA  (SP),Y      ; low byte
C110  85 FB          STA  ZL
C112  18             CLC              ; clear carry means OK
C113  60             RTS
C114  38        ABORT SEC             ; set carry means not OK
C115  60             RTS
                                      ;
C116  0E EC C1  LEFTZL ASL MASK       ; move the bit in MASK to the left
C119  90 1A          BCC  LEFTOK      ; within the byte, it's OK
C11B  6E EC C1       ROR  MASK        ; put the bit back in position 7, just in case
                                      ; Better check for a left edge.
C11E  20 45 C1       JSR  CHECKEDGE
C121  90 01          BCC  DECZL       ; carry clear means OK
C123  60             RTS              ; else, return because we've hit the left
                                      ; edge of the screen
C124  A5 FB     DECZL LDA ZL          ; subtract
C126  E9 07          SBC  #7          ; 7 (really subtract 8, because carry is clear)
C128  85 FB          STA  ZL          ; store it
C12A  A5 FC          LDA  ZL+1        ; high byte
C12C  E9 00          SBC  #0          ; adjust
C12E  85 FC          STA  ZL+1        , and store
C130  A9 01          LDA  #1          ; and put a %00000001
C132  8D EC C1       STA  MASK        ; into mask
C135            LEFTOK =  *           ; now check the bit
C135  A0 00          LDY  #0
C137  B1 FB          LDA  (ZL),Y      ; get the byte
C139  4D F4 C1       [illegible] FINDL ; flip the bits to get what we're looking for
C13C  2D EC C1       AND  MASK        ; check the bitmap bit
C13F  F0 D5          BEQ  LEFTZL      ; if zero, do more
C141  20 6B C1       JSR  RIGHTZL     ; else move ZL to the right
C144  60             RTS              ; and quit
                                      ;
C145  A5 FB     CHECKEDGE LDA ZL      ; low byte
C147  AA             TAX              ; save it
C148  29 38          AND  #%00111000  ; check bits 3-5
C14A  D0 1B          BNE  NOPROB      ; no problem, we're done
C14C  8A             TXA              ; check more
C14D  29 C0          AND  #%11000000  ; get the two high bits
C14F  8D F7 C1       STA  TEMP        ; and save in temp
C152  A5 FC          LDA  ZL+1        ; high byte
C154  29 1F          AND  #%00011111  ; mask off the three high bits
C156  2E F7 C1       ROL  TEMP        ; move into .A
C159  2A             ROL
C15A  2E F7 C1       ROL  TEMP        ; two bits
C15D  2A             ROL
C15E  F0 09          BEQ  PROB        , see if it's divisible by five
C160  38             SEC
```

```
C161  E9 05        DUNNO    SBC  #5            ; subtract 5
C163  F0 04                 BEQ  PROB          ; zero means a problem
C165  B0 FA                 BCS  DUNNO         ; carry set means more
C167  18           NOPROB   CLC                ; no problem
C168  60                    RTS
C169  38           PROB     SEC                ; problem/left edge
C16A  60                    RTS
                                               ;
C16B               RIGHTZL  =    *             ; this routine moves ZL right one pixel
C16B  4E EC C1              LSR  MASK
C16E  B0 01                 BCS  RTEDGE        ; if the bit rotated into the carry flag, check
                                               ; for the edge
C170  60                    RTS                ; else, it's OK
C171  A5 FB        RTEDGE   LDA  ZL            ; low byte
C173  69 07                 ADC  #7            ; really add 8—carry is set
C175  85 FB                 STA  ZL            ; store it
C177  90 02                 BCC  SKPHI         ; skip the high byte
C179  E6 FC                 INC  ZL+1          ; unless ZL has overflowed
C17B  20 45 C1     SKPHI    JSR  CHECKEDGE     ; see if we're at an edge
C17E  90 13                 BCC  RTOK          ; it's OK
C180  A5 FB                 LDA  ZL            ; or not OK
C182  E9 08                 SBC  #8            ; implied carry set
C184  85 FB                 STA  ZL            ; subtract 8 from low byte
C186  A5 FC                 LDA  ZL+1          ; plus
C188  E9 00                 SBC  #0            ; maybe
C18A  85 FC                 STA  ZL+1          ; the high byte
C18C  A9 01                 LDA  #%00000001    ; and put the one bit
C18E  8D EC C1              STA  MASK          ; at the edge of mask
C191  38                    SEC                ; set carry means finish
C192  60                    RTS                ; this ends the RIGHTZL routine
C193  A9 80        RTOK     LDA  #%10000000    ; set up mask
C195  8D EC C1              STA  MASK
C198  18                    CLC                ; clear carry signals all is well
C199  60                    RTS                ; end on a positive note
                                               ;
C19A               SETZUZD  =    *             ; first set ZU (the pointer to the pixel
                                               ; above ZL)
C19A  A5 FC                 LDA  ZL+1          ; high byte
C19C  A8                    TAY
C19D  85 FA                 STA  ZU+1
C19F  85 FE                 STA  ZD+1          ; and ZD
C1A1  A5 FB                 LDA  ZL            ; low byte
C1A3  AA                    TAX                ; save in .X
C1A4  85 F9                 STA  ZU
C1A6  85 FD                 STA  ZD
C1A8  29 07                 AND  #7            ; check for eight-byte edge, top or bottom
C1AA  F0 09                 BEQ  FIXZU         ; if ZL is divisible by 8
C1AC  C9 07                 CMP  #7
C1AE  F0 1C                 BEQ  FIXZD         ; or one less than 8
C1B0  C6 F9                 DEC  ZU            ; else ZU is one less
C1B2  E6 FD                 INC  ZD            ; and ZD is one more
C1B4  60                    RTS
C1B5  E6 FD        FIXZU    INC  ZD            ; ZD is OK. INC it.
C1B7  8A                    TXA
C1B8  38                    SEC
C1B9  E9 39                 SBC  #<313         ; move back a line
C1BB  85 F9                 STA  ZU
C1BD  98                    TYA                ; high byte
C1BE  E9 01                 SBC  #>313
C1C0  85 FA                 STA  ZU+1
C1C2  CD F1 C1              CMP  BITMAP+1      ; check if it's too low
C1C5  B0 04                 BCS  FXZOK         ; no, go to the end
```

```
C1C7  86  F9            STX    ZU           ; too low, put ZL into ZU
C1C9  84  FA            STY    ZU+1
C1CB  60        FXZOK   RTS
                                            ;
C1CC  C6  F9    FIXZD   DEC    ZU           ; ZU is OK. DEC it.
C1CE  8A                TXA                 ; low byte of ZL/ZD
C1CF  18                CLC
C1D0  69  39            ADC    #<313        ; move up a line
C1D2  85  FD            STA    ZD
C1D4  98                TYA                 ; high byte
C1D5  69  01            ADC    #>313
C1D7  85  FE            STA    ZD+1
C1D9  CD  F3  C1        CMP    BITMAX+1     ; check if it's too high
C1DC  90  0D            BCC    FXDOK        ; no, go to the end
C1DE  D0  07            BNE    TOOHI        ; if carry is set and it's not equal, it's too
                                            ; high
                                            ; It's equal, so check the low byte.
C1E0  AD  F2  C1        LDA    BITMAX
C1E3  C5  FD            CMP    ZD
C1E5  B0  04            BCS    FXDOK        ; if BITMAX >= ZD, don't worry, else drop
                                            ; through
C1E7  86  FD    TOOHI   STX    ZD           ; too high, put ZL into ZU
C1E9  84  FE            STY    ZD+1
C1EB  60        FXDOK   RTS
                                            ;
C1EC  00        MASK    .BYTE  0            ; mask for turning bits on/off
C1ED  A0  00    STARTX  .WORD 160           ; starting location for fill (x-position =
                                            ; 0-319)
C1EF  65        STARTY  .BYTE  101          ; starting location (y-position = 0 199)
C1F0  00  20    BITMAP  .WORD 8192          ; start of the bitmap, $2000
C1F2  3F  3F    BITMAX  .WORD 16191
C1F4  00        FINDL   .BYTE  0            ; set to zero if changing zeros to ones, or 255
                                            ; if 1 to 0
C1F5  00        FINDUP  .BYTE  0
C1F6  00        FINDDN  .BYTE  0
C1F7  00        TEMP    .BYTE  0
C1F8            STACK   =      *
```

See also BITMAP, CLRHRF, CLRHRS, HRCOLF, HRPOLR, HRSETP.

Name

Pass values from BASIC to ML using the FRMEVL routine

Description

This is the most versatile of the techniques that pass a value from BASIC to ML.

Prototype

1. Call the COMMA routine to find a comma.
2. Call the FRMEVL routine to calculate the value between commas (the result is stored in the floating-point accumulator).
3. Use the number as you wish.

Explanation

FRMEVL evaluates a formula by calling various BASIC functions and stripping away the parentheses. FRMEVL can figure out what ABS(INT(Y/2)) + SQR(X * 2 − Z + 3) really means.

The example routine adds two integers. You pass the values to the ML routine by adding commas and formulas after the SYS. For example, SYS 49152,1,2 will print the number 3; SYS 49152,SQR(9),(1 + 3*7) will print the number 25 (3 + 22).

The three key ROM routines are COMMA, which looks for the next comma; FRMEVL, which evaluates the formula; and QINT, which converts a floating-point number to an integer.

Routine

```
C000                HI       =    100        ; high byte after QINT
C000                LO       =    101        ; low byte
C000                COMMA    =    $AEFD      ; routine that looks for a comma
C000                FRMEVL   =    $AD9E      ; evaluate expression
C000                QINT     =    $BC9B      ; convert floating-point number in FAC1 to
                                             ; integer
C000                LINPRT   =    $BDCD      ; print an integer
                                             ;
C000  20  FD  AE    PASFMV   JSR  COMMA      ; look for a comma
C003  20  9E  AD             JSR  FRMEVL     ; evaluate the expression
C006  20  9B  BC             JSR  QINT       ; convert the FP number to a four-byte
                                             ; integer
C009  A5  65                 LDA  LO         ; low byte
C00B  8D  32  C0             STA  TOTAL      ; store it
C00E  A5  64                 LDA  HI         ; high byte
C010  8D  33  C0             STA  TOTAL+1    ; is saved also
                                             ;
C013  20  FD  AE             JSR  COMMA      ; get the next number
C016  20  9E  AD             JSR  FRMEVL     ; and figure it out
C019  20  9B  BC             JSR  QINT       ; convert
C01C  18                     CLC
C01D  A5  65                 LDA  LO         ; get the low byte
C01F  6D  32  C0             ADC  TOTAL      ; add it
```

```
C022  8D 32 C0              STA   TOTAL
C025  AA                    TAX                 ; place it in .X
C026  A5 64                 LDA   HI            ; add in
C028  6D 33 C0              ADC   TOTAL+1       ; the high byte, also
C02B  8D 33 C0              STA   TOTAL+1
C02E  20 CD BD              JSR   LINPRT        ; print the number
C031  60                    RTS
                                                ;
C032  00 00       TOTAL     .BYTE 0,0
```

See also GOTOBL, PASMEM, PASREG, PASUSR.

Name

Pass values from BASIC to ML by POKEing to free memory

Description

Although this technique limits the values you can pass to numbers in the range 0–255, it's one of the simplest ways to pass numbers back and forth from BASIC to ML. Use PEEK and POKE in BASIC, LDA and STA in ML.

Prototype

1. In BASIC, POKE a value to a free memory location. Then SYS to the machine language routine.
2. In the ML program, LDA (or LDX or LDY) the number and handle it as you wish.

Explanation

The example is relatively simple. In BASIC, POKE 828 with a number 0–255, then SYS 49152. A delay loop, based on the number in location 828, will execute (MEM, in the example). The maximum delay is 255 jiffies, or about four seconds. While the delay loop is running, the border color flashes very quickly.

Routine

```
C000              JIF     =    $A2       ; low byte of jiffy clock (both 64 and 128)
C000              MEM     =    828       ; free RAM in the cassette buffer for the 64;
                                         ; use another free memory location on the
                                         ; 128
C000              BORCOL  =    53280     ; border color register
                                         ;
C000  78          PASMEM  SEI            ; turn off interrupts while the routine is set
                                         ; up
C001  AD 20 D0            LDA  BORCOL    ; get the border color
C004  8D 21 C0            STA  TEMP      ; save it
C007  AD 3C 03            LDA  MEM       ; get the value
C00A  F0 0E               BEQ  QUIT      ; if the delay is zero, don't do anything
C00C  18                  CLC            ; prepare to add
C00D  65 A2               ADC  JIF       ; to the current jiffy value
C00F  58                  CLI            ; interrupts now on
                                         ;
C010  C5 A2       LOOP    CMP  JIF       ; compare .A to the clock
C012  F0 06               BEQ  QUIT      ; if they're equal, end the delay
C014  EE 20 D0            INC  BORCOL    ; flashing effect for the border
C017  4C 10 C0            JMP  LOOP      ; go back
                                         ;
C01A  AD 21 C0    QUIT    LDA  TEMP
C01D  8D 20 D0            STA  BORCOL    ; restore the border color
C020  60                  RTS            ; and end
                                         ;
C021  00          TEMP    .BYTE 0
```

See also GOTOBL, PASFMV, PASREG, PASUSR.

PASREG

Name

Pass values to an ML program directly through the registers

Description

By POKEing to locations 780–783 (64 only), you can set the values that the registers .A, .X, .Y, and the processor status .P, respectively, will hold at the beginning of a routine called with the BASIC statement SYS. BASIC itself handles the task of transferring the contents of these locations into the proper registers. An equivalent technique for the 128 is simply to include the desired values, separated by commas, following the SYS address.

Prototype

1. Before SYSing to the routine, POKE the desired register values into 780–783.
2. In the routine, handle the values as needed.

Explanation

The example routine saves .A, clears the carry flag, JSRs to the Kernal PLOT routine, and then prints the character in .A. To call it from BASIC, assuming you want to print the letter C at row 20, column 3, use this syntax:

POKE 780,67: POKE 781,19: POKE 782,2: SYS 49152 Commodore 64
SYS 3072,67,19,2 Commodore 128

The 64 routine is at 49152, and the 128 routine is at 3072. After returning from the ML program, you can find the previous values of .A, .X, .Y, and .P by PEEKing locations 780–783 on the 64, or by using RREG, the Read REGister statement, on the 128.

Routine

```
C000                 CHROUT  =    $FFD2
C000                 PLOT    =    $FFF0
                                             ;
C000                 PASREG  =    *          ; .A, .X, and .Y should already hold values
C000  48                     PHA             ; save .A, because plot might affect it
C001  18                     CLC             ; get ready to plot
C002  20  F0  FF             JSR  PLOT       ; x and y position are set
C005  68                     PLA             ; get .A back
C006  20  D2  FF             JSR  CHROUT     ; print it
C009  60                     RTS             ; quit
```

See also GOTOBL, PASFMV, PASMEM, PASUSR.

Name

Pass values from BASIC to ML via the USR function

Description

In BASIC, you can include a line such as X = USR(G), where the value of the variable G is sent (as a floating-point value) to a machine language routine stored in memory. The ML routine can then pass another floating-point value back to the BASIC program, where it will be assigned to the variable X.

Prototype

1. Set up the USR function by POKEing the address of your ML routine into locations 785–786 (locations 4633–4634 on the 128).
2. Calculate, transform, or otherwise use the value in the floating-point accumulator.

Explanation

The example routine takes three values. If the value passed is 1, the screen is cleared. If it's 2, the cursor color is changed to white. If it's 3, the string *HELLO* is printed. The QINT BASIC ROM routine converts the floating-point value to an integer to be handled by the ML program.

After assembling the program to 49152, **POKE 785,0:POKE786,192** to set up the pointer. On the 128, substitute **POKE 4633,0: POKE 4634,12** (these are the low and high bytes of $0C00). You'll also need to change HI and LO to 102 and 103, and QINT to $8CC7 on the 128. Use **PASUSR** from BASIC with a statement of the form **Z = USR(1)** or USR(2) or USR(3).

Routine

```
C000                HI       =     100        ; HI = 102 on the 128—high byte after
                                              ; QINT
C000                LO       =     101        ; LO = 103 on the 128—low byte
C000                QINT     =     $BC9B      ; QINT = $8CC7 on the 128—convert
                                              ; floating point number in FAC1 to integer
C000                CHROUT   =     $FFD2      ; Kernal print routine
                                              ;
C000  20 9B BC      PASUSR   JSR   QINT       ; convert FAC1 to an integer
C003  A6 65                  LDX   LO         ; get the low byte
C005  F0 1B                  BEQ   DONE       ; if zero, quit
C007  E0 04                  CPX   #4         ; if it's greater than 3
C009  B0 17                  BCS   DONE       ; skip ahead and quit
C00B  CA                     DEX              ; count down 3-2-1
C00C  D0 05                  BNE   MORE1      ; if it was 2 or 3, keep going
C00E  A9 93         FN1      LDA   #147       ; clear screen
C010  4C D2 FF               JMP   CHROUT     ; print it (implied RTS)
                                              ;
C013  CA            MORE1    DEX              ; count down again
```

```
C014  D0 05           BNE  MORE2     ; if not zero, move ahead
C016  A9 05       FN2 LDA  #5        ; code for <white>
C018  4C D2 FF        JMP  CHROUT    ; print it (RTS built in)
C01B  A0 00     MORE2 LDY  #0
C01D  B9 2A C0  ULOOP LDA  GREET,Y   ; get a character
C020  D0 01           BNE  PRINIT    ; if zero
C022  60         DONE RTS            ; then quit
                                     ;
C023  20 D2 FF PRINIT JSR  CHROUT    ; else print it
C026  C8              INY            ; loop counts forward
C027  4C 1D C0        JMP  ULOOP     ; and go back for more
                                     ;
C02A  48 45 4C  GREET .ASC "HELLO"
C02F  0D 00           .BYTE 13,0
```

See also GOTOBL, PASFMV, PASMEM, PASREG.

Name

Set the cursor location

Description

PLOTCR lets you locate characters anywhere on the screen without requiring you to use the cursor characters. It relies on the Kernal routine PLOT to position the cursor for subsequent printing.

Prototype

1. Enter this routine with the desired cursor position in .X (row) and .Y (column).
2. Clear the carry flag.
3. JSR to the Kernal routine PLOT and RTS (or simply JMP to PLOT).

Explanation

In the example program, the cursor is positioned in the fifth column of the fourth row, and an *E* is printed.

The X register should contain the appropriate row number minus one, while .Y contains the column number less one. If you are working within a window on the 128, the row and column values are relative to the top and left sides of the window rather than to the screen borders.

Note: Using .X for the row and .Y for the column is backward from what you might think. In most Cartesian coordinate systems, *x* is the horizontal axis (columns) and *y* is the vertical axis (rows). The Kernal PLOT routine is just the opposite.

Warning: Be sure to clear the carry flag before accessing PLOT. Otherwise, if carry is set, PLOT will return the current cursor position in .X and .Y (used in **FINDCR**).

Routine

```
C000        PLOT    =   65520     ; Kernal cursor position routine
C000        CHROUT  =   65490
                                  ;
                                  ; Print an E at (4,5).
```

```
C000  A9 93        CLRCHR  LDA  #147     ; clear the screen
C002  20 D2 FF             JSR  CHROUT
C005  A2 03                LDX  #3       ; fourth row
C007  A0 04                LDY  #4       ; fifth column
C009  20 12 C0             JSR  PLOTCR   ; position the cursor
C00C  A9 45                LDA  #69      ; print E
C00E  20 D2 FF             JSR  CHROUT
C011  60                   RTS
                                         ;
                                         ; Position the cursor at ( Y, X)
C012  18           PLOTCR  CLC           ; clear carry to set position
C013  20 F0 FF             JSR  PLOT     ; position cursor
C016  60                   RTS
```

See also FINDCR.

Name

POKE RAM under ROM / PEEK RAM under ROM

Description

When you turn on the 64, the 8K BASIC interpreter ROM at 40960 and the 8K operating system Kernal ROM at 57344 are selected. But under both of these 8K areas is free RAM which you can access by altering the contents of the memory configuration register at location 1.

These areas of free memory can be used in many ways: You can store your ML programs there so that they are in visible to BASIC, or you can use the space as a data storage area for disk copying, word processing, and sorting routines.

With the aid of two routines, **POKRUR** and **PEKRUR**, the example program demonstrates how the area of memory under BASIC ROM can be used as a buffer for storing the first two screen lines.

Prototype

In **POKRUR**:

1. In the subroutine TXTPTR, store the address of memory to be transferred (or the origin address) in ZP; store the address of the target buffer in RAM under ROM in ZP+2.
2. Using the subroutine NUMMOV, store the number of bytes to transfer (defined as NUMBER at the end of the program) in BUFCTR.
3. With the subroutine MOVEIT, transfer memory from the origin address in ZP to the target address in ZP+2.

In **PEKRUR**:

1. Push the current RAM/ROM configuration register in location 1 on the stack.
2. Select in RAM under BASIC ROM at 40960.
3. Using the subroutine TXTPTR, store the address of the buffer in RAM under ROM in ZP, and the destination address of regular RAM in ZP+2.
4. Fetch the number of bytes to move (NUMBER) and store this value in BUFCTR with NUMMOV.
5. With MOVEIT, transfer memory from the address in ZP to the address in ZP+2.
6. Restore the RAM/ROM configuration register in location 1.

Explanation

If you POKE into the 8K of memory at 40960 or at 57444, whatever you POKE is always stored into the underlying RAM. PEEKing these areas of memory, on the other hand, will return either the contents of ROM or of the RAM underneath, depending on the state of the configuration register. These principles are illustrated by **POKRUR** and **PEKRUR** in the program that follows.

The program inserts an IRQ interrupt routine that allows you to save or retrieve the first two screen lines (text only) placed in a buffer area at 40960. The IRQ routine WEDGE checks for two keys, F1 and F3.

If the user presses F1, **POKRUR** saves text from the top two screen lines. A border color change indicates a successful save. When F3 is pressed, **PEKRUR** recalls these lines.

POKRUR and **PEKRUR** have three subroutines in common: TXTPTR, NUMMOV, and MOVEIT. Zero-page addressing is used in MOVEIT to transfer bytes from the screen to the buffer or vice versa. In this subroutine, memory is always moved from the address in ZP, or the origin address, to the address in ZP+2, or the destination address.

And this is where TXTPTR comes into play. This subroutine sets the zero-page pointers according to the direction of the move. In order to do this, a 0 or a 2 must be in the X register. If you're performing a save (.X = 0), TXTPTR initially points ZP to TEXT at 1024, and ZP+2 to BUFFER at 40960. Conversely, if you're retrieving the buffer (.X = 2), it points ZP to BUFFER and ZP+2 to TEXT.

The third subroutine, NUMMOV, takes the number of bytes to move—in this case, 80—from NUMBER and stores this value in a counter (BUFCTR) used by MOVEIT.

There's little more to **POKRUR** than these three subroutines. After the text is stored, exit the routine through the normal IRQ interrupt handler.

PEKRUR is slightly more involved. Before fetching the two screen lines in the buffer, save the contents of the configuration register at location 1 so that you can later restore it. Next, select RAM under ROM at 40960 by turning off bit 0 in location 1, and execute the three subroutines (TXTPTR, NUMMOV, and MOVEIT).

To finish the routine, restore the memory configuration register and again exit through the interrupt service routine.

Note: Follow the same procedure in accessing RAM under Kernal ROM. The only difference is that you flip bit 1 rather than bit 0 in location 1. Also, since the interrupt-service routine is handled in the Kernal area, you must turn off interrupts with SEI before you access the RAM under ROM.

Routine

```
C000                 ZP       =    251
C000                 IRQVEC   =    788          ; vector to IRQ interrupt routine
C000                 IRQNOR   =    59953        ; normal IRQ interrupt service routine
C000                 LSTX     =    197          ; last key pressed
C000                 EXTCOL   =    53280        ; border color register
C000                 TEXT     =    1024         ; location of text to be stored
C000                 BUFFER   =    40960        ; text storage buffer under BASIC ROM
                                                ;
                                                ; Insert IRQ interrupt wedge to store the top
                                                ; two screen lines in RAM
                                                ; under BASIC ROM with F1 key F3 brings
                                                ; back the two lines in the buffer.
C000  78             SETUP    SEI               ; disable IRQ interrupts to change IRQ vector
                                                ; Then store the address of our routine into
                                                ; IRQ vector.
C001  A9 0D                   LDA  #<WEDGE      ; low byte first
C003  8D 14 03                STA  IRQVEC
C006  A9 C0                   LDA  #>WEDGE      ; then high byte
C008  8D 15 03                STA  IRQVEC+1
C00B  58                      CLI               ; We've reset the vector. Now reenable IRQ
                                                ; interrupts and
C00C  60                      RTS               ; exit setup.
                                                ;
C00D  A5 C5          WEDGE    LDA  LSTX         ; fetch the last keypress
C00F  C9 04                   CMP  #4           ; is it F1?
C011  F0 07                   BEQ  POKRUR       ; save the top two screen lines
C013  C9 05                   CMP  #5           ; is it F3?
C015  F0 14                   BEQ  PEKRUR       ; recall the two screen lines
C017  4C 31 EA       EXIT     JMP  IRQNOR       ; service the standard IRQ routines
                                                ;
                                                ; POKRUR stores the number of bytes in
                                                ; number to RAM under BASIC ROM
C01A  EE 20 D0       POKRUR   INC  EXTCOL       ; change border color to indicate buffer
                                                ; storage
C01D  A2 00                   LDX  #0           ; so ZP points to TEXT (origin), ZP+2 to
                                                ; BUFFER (target)
C01F  20 43 C0                JSR  TXTPTR       ; set up zero-page pointers to TEXT and
                                                ; BUFFER
C022  20 5B C0                JSR  NUMMOV       ; get number of bytes to move
C025  20 68 C0                JSR  MOVEIT       ; store TEXT in BUFFER
C028  4C 17 C0                JMP  EXIT         ; to standard IRQ interrupt routines
                                                ;
                                                ; PEKRUR gets the number of bytes in
                                                ; number from RAM under BASIC ROM.
C02B  A5 01          PEKRUR   LDA  1            ; store the current RAM/ROM
                                                ; configuration on the stack
C02D  48                      PHA
C02E  29 FE                   AND  #%11111110   ; select RAM under BASIC ROM by
                                                ; turning off bit 0
C030  85 01                   STA  1
C032  A2 02                   LDX  #2           ; so two bytes at ZP point to BUFFER
                                                ; (origin), ZP+2 to TEXT (target)
C034  20 43 C0                JSR  TXTPTR       ; set up zero-page pointers to BUFFER and
                                                ; TEXT
```

```
C037  20  5B  C0            JSR   NUMMOV       ; fetch the number of bytes to move
C03A  20  68  C0            JSR   MOVEIT       ; recall TEXT from BUFFER
C03D  68                    PLA                ; restore RAM/ROM configuration
C03E  85  01                STA   1
C040  4C  17  C0            JMP   EXIT         ; take care of normal IRQ routines
                                               ;
                                               ; Set origin and target pointers. Enter with
                                               ; .X = 0 to point ZP to TEXT,
                                               ; ZP+2 to BUFFER. Enter with .X = 2 to
                                               ; point ZP to BUFFER, ZP+2 to TEXT
C043  A9  00      TXTPTR    LDA   #<TEXT       ; get low byte of TEXT
C045  95  FB                STA   ZP,X         ; store to ZP (if .X was 0) or ZP+2 (if .X
                                               ; was 2)
C047  E8                    INX                ; for high byte
C048  A9  04                LDA   #>TEXT       ; get high byte of TEXT
C04A  95  FB                STA   ZP,X         ; store to ZP+1 (if .X was 0) or ZP+3 (if .X
                                               ; was 2)
C04C  CA                    DEX                ; set index back to 0 (if .X was 0) or 2 (if .X
                                               ; was 2)
C04D  8A                    TXA
C04E  49  02                EOR   #2           ; change .X from 0 to 2 or vice versa
C050  AA                    TAX
C051  A9  00                LDA   #<BUFFER     ; get low byte of BUFFER
C053  95  FB                STA   ZP,X         ; store to ZP+2 (if .X was 0) or ZP (if .X
                                               ; was 2)
C055  E8                    INX                ; for high byte
C056  A9  A0                LDA   #>BUFFER     ; get high byte of buffer
C058  95  FB                STA   ZP,X         ; store to ZP+3 (if .X was 0) or ZP+1 (if .X
                                               ; was 2)
C05A  60                    RTS
                                               ;
                                               ; Store number of bytes to transfer in
                                               ; BUFCTR.
C05B  AD  8A  C0  NUMMOV    LDA   NUMBER       ; low byte first
C05E  8D  8C  C0            STA   BUFCTR
C061  AE  8B  C0            LDX   NUMBER+1     ; then high byte
C064  8E  8D  C0            STX   BUFCTR+1
C067  60                    RTS
                                               ;
                                               ; MOVEIT moves bytes from address in ZP to
                                               ; address in ZP+2.
C068  A0  00      MOVEIT    LDY   #0           ; as an index in MOVELP
C06A  B1  FB      MOVELP    LDA   (ZP),Y       ; get a byte from origin (TEXT or BUFFER)
C06C  91  FD                STA   (ZP+2),Y     ; and move it
                                               ; Increment zero-page pointers for both origin
                                               ; and target.
                                               ; Increment the origin ZP pointers first.
C06E  E6  FB                INC   ZP           ; increment low byte
C070  D0  02                BNE   INCTAR       ; if low byte hasn't turned over, increment
                                               ; target pointers
C072  E6  FC                INC   ZP+1         ; increment high byte
C074  E6  FD      INCTAR    INC   ZP+2         ; increment the low byte of the target pointer
C076  D0  02                BNE   LENCHK       ; if low byte hasn't turned over, skip over
                                               ; high-byte increment
```

```
C078  E6  FE      INCZP2   INC    ZP+3        ; increment high byte of target pointer
C07A  CE  8C  C0  LENCHK   DEC    BUFCTR      ; decrement low byte of buffer counter
C07D  D0  EB               BNE    MOVELP      ; if not equal, more of the buffer remains, so
                                              ; continue moving
C07F  CE  8D  C0           DEC    BUFCTR+1    ; otherwise, decrement high byte of buffer
                                              ; counter
C082  AD  8D  C0           LDA    BUFCTR+1    ; continue moving until last page of buffer
                                              ; has transferred
C085  C9  FF               CMP    #255        ; high byte goes from 0 through 255 on last
                                              ; page
C087  D0  E1               BNE    MOVELP      ; we've yet to reach last page, so continue
                                              ; moving
C089  60                   RTS
                                              ;
C08A  50  00      NUMBER   .WORD 80           ; number of bytes to transfer
C08C  00  00      BUFCTR   .WORD 0            ; two-byte counter for remaining number of
                                              ; bytes to move
```

Name

POKE to screen and color memory

Description

With **POKSCR**, you can position a series of colored characters beginning at any location on the text screen.

Prototype

1. Define the screen codes of the characters you want to place on the screen (as SCODE) and their corresponding color values (as COLVAL).
2. Set SCREEN equal to the first screen position where the characters will be placed.
3. Load the accumulator with the low byte of SCREEN and .X with its high byte. Then JSR to LOCATE.
4. In LOCATE, store the starting text position in zero page. Calculate the starting color-RAM position and store it in zero page as well.
5. Using zero-page addressing, store the screen codes in text memory and colors in color RAM.

Explanation

The following program puts the message LINE 3 at the beginning of line 3 on the text screen. Each character within the message is shown in a different color (except for the space).

The subroutine LOCATE puts the initial text position (SCREEN) and color-RAM position for the message in zero page. The proper color memory address is determined by performing a two-byte addition of SCREEN to OFFSET, where OFFSET represents the difference between text and color memory.

POKSCR can easily be modified to store screen codes elsewhere in screen memory. Put the list of screen codes for your characters in SCODE and the color of each in COLVAL. Change SCREEN to the desired screen location. Then count the number of screen codes and replace the six within POKELP with this number.

For a table of color values, see **COLFIL**.

Routine

```
C000          OFFSET   =    54272     ; offset to color RAM
C000          SCREEN   =    1104      ; starting screen position where characters are
                                      ; stored
C000          ZP       =    251
                                      ;
```

```
                                               ; Store screen codes to memory with color.
C000  A9 50      POKSCR  LDA   #<SCREEN        ; low byte of screen position
C002  A2 04              LDX   #>SCREEN        ; and high byte
C004  [illegible] 19 C0  JSR   LOCATE          ; put screen position, text and color RAM,
                                               ; in zero page
                                               ; Now place characters in screen memory in
                                               ; color.
C007  A0 00              LDY   #0              ; as an index
C009  B9 2E C0   POKELP  LDA   COLVAL,Y
C00C  91 FD              STA   (ZP+2),Y        ; store the color for character in SCODE
                                               ; plus .Y
C00E  B9 [illegible] C0  LDA   SCODE,Y
C011  91 FB              STA   (ZP),Y          ; store each screen code
C013  C8                 INY                   ; next screen code
C014  C0 06              CPY   #6              ; have we done all six?
C016  D0 F1              BNE   POKELP          ; if not, continue
C018  60                 RTS
                                               ;
                                               ; Enter with low (.A) and high (.X) bytes of
                                               ; screen position.
                                               ; Store starting text position in ZP and
                                               ; ZP+1, color in ZP+2 and ZP+3.
C019  85 FB      LOCATE  STA   ZP              ; store screen first position
C01B  86 FC              STX   ZP+1
                                               ; Add in offset for color memory.
C01D  18                 CLC
C01E  69 00              ADC   #<OFFSET        ; low byte first
C020  [illegible] FD     STA   ZP+2
C022  [illegible]        TXA
C023  69 D4              ADC   #>OFFSET        ; then high byte
C025  85 FE              STA   ZP+3
C027  60                 RTS
                                               ;
C028  0C 09 0E   SCODE   .BYTE 12,9,14,5,32,51
                                               ; screen codes for "LINE 3"
C02E  05 02 07   COLVAL  .BYTE 5,2,7,4,4,14
                                               ; colors—GRN, RED, YEL, PUR, PUR, LT
                                               ; BLU
```

See also PRTCHR.

Name

Print a character on the screen

Description

You'll need this routine anytime you print a character on the text screen. **PRTCHR** relies on the Kernal routine CHROUT to locate a character at the current cursor position.

Prototype

1. Enter this routine with the ASCII value of the character you want to print in .A (defined as CHAR).
2. JSR to the Kernal routine CHROUT and RTS (or simply JMP to CHROUT).

Explanation

The example program clears the screen with **CLRCHR** and prints a *J*.

Note: On the 128, CHROUT is also referred to as BSOUT.

Routine

```
C000               CHROUT  =     65490     ; Kernal character output routine
                                           ;
                                           ; Clear screen and print J.
C000  A9 93        CLRCHR  LDA   #147      ; clear the screen
C002  20 D2 FF             JSR   CHROUT
C005  AD 10 C0             LDA   CHAR      ; get the character
C008  20 0C C0             JSR   PRTCHR    ; and print it
C00B  60                   RTS
                                           ;
                                           ; Print the character in .A.
C00C  20 D2 FF     PRTCHR  JSR   CHROUT    ; print it at the current cursor location
C00F  60                   RTS
C010  4A           CHAR    .BYTE 74        ; ASCII value for J
```

See also POKSCR.

Name

Send characters to the printer

Description

Open a channel to the printer and output an ASCII character

Prototype

1. Using **OPENPR**, open the printer channel with the parameters 4,4,0.
2. Load the accumulator with the ASCII character you wish to print.
3. Print it with the Kernal routine CHROUT.
4. With the file number in .A, JMP to **CLOSFL** to close the printer channel and restore output to the screen.

Explanation

The example program opens the printer as channel 4 and prints an uppercase *T*. For a program that prints an entire string, see **PRTSTR**.

Note: For most printers, the logical file number for the output can be any integer in the range 0–255; the device number is usually 4. Some printers can also use 5 as a device number.

The secondary address sends information on Commodore printers about the character set. A value of 0 causes Commodore printers to print in uppercase and graphics. A value of 7 causes them to print in uppercase and lowercase. Some printers require a value of 255 (for no secondary address) here. Consult your printer or interface manual to determine the exact significance these parameters have with your printer or printer interface.

Finally, the last couple of instructions are necessary on certain printers that store output in a buffer before printing it. Printing the carriage return insures that this buffer gets printed.

Routine

```
C000                SETLFS   =    65466
C000                OPEN     =    65472
C000                CHKOUT   =    65481
C000                CHROUT   =    65490
C000                CLOSE    =    65475
C000                CLRCHN   =    65484
                                          ; Open a file to the printer with OPENPR,
                                          ; print T, and
                                          ; close printer channel with CLOSFL.
C000  20  12  C0  PRTOUT   JSR   OPENPR   ; open the printer
C003  A9  54               LDA   #84      ; print T
C005  20  D2  FF           JSR   CHROUT
```

```
C008  A9 0D               LDA  #13      ; print RETURN to clear printer buffer
C00A  20 D2 FF            JSR  CHROUT
C00D  A9 04               LDA  #4       ; file to close
C00F  4C 23 C0            JMP  CLOSFL   ; close file to printer, restore
                          ;
                                        ; OPEN the printer as 4,4,0.
C012  A9 04      OPENPR   LDA  #4       ; logical file 4
C014  A2 04               LDX  #4       ; device number (printer is usually device 4,
                                        ; sometimes 5)
C016  A0 00               LDY  #0       ; secondary address of 0
                                        ; A value of 0 here causes Commodore
                                        ; printers to print in uppercase/graphics.
                                        ; A value of 7 causes Commodore printers to
                                        ; print in lowercase/uppercase.
                                        ; Some printers require a value of 255
                                        ; (meaning no secondary address).
                                        ;
C018  20 BA FF            JSR  SETLFS   ; set values
C01B  20 C0 FF            JSR  OPEN     ; open a file to printer (OPEN 4,4,0)
C01E  A2 04               LDX  #4
C020  4C C9 FF            JMP  CHKOUT   ; direct output to file 4 (that is, CMD 4) and
                                        ; RTS
                                        ;
                                        ; CLOSFL closes the logical file in .A and
                                        ; restores default devices.
C023  20 C3 FF   CLOSFL   JSR  CLOSE    ; close file in .A
C026  4C CC FF            JMP  CLRCHN   ; clear all channels, restore default devices
                                        ; and RTS
```

See also CLOSFL, OPENPR, PRTSTR.

Name

Send a string to the printer

Description

PRTSTR opens a channel to the printer and prints an ASCII string.

Prototype

1. OPEN the printer channel with the parameters 4,4,0 by using **OPENPR**.
2. JSR to a string-printing routine.
3. After printing the string, send a carriage return to clear the printer buffer.
4. With the number of the open file in .A, JMP to **CLOSFL** to close the printer channel and restore output to the screen.

Explanation

The example program opens the printer as channel 4 and prints HELLO.

Notice the custom printing routine **STRCPT**; it works with both the 64 and the 128. You could shorten the program somewhat by substituting **STP64** on the 64 or **STP128** on the 128.

To print individual characters, see **PRTOUT**.

Note: For most printers, the logical file number for the output can be any integer in the range 0–255. The device number is usually 4 (with nearly all Commodore printers). Some printers can also use 5 as a device number.

The secondary address sends information on Commodore printers about the character set. A value of 0 causes Commodore printers to print in uppercase and graphics. A value of 7 causes them to print in uppercase and lowercase. Some printers require a value of 255 (for no secondary address) here. It is best to consult your printer manual to determine the exact significance that these parameters will have with your printer and/or interface.

Routine

```
C000            SETLFS  =   65466
C000            OPEN    =   65472
C000            CHKOUT  =   65481
C000            CHROUT  =   65490
C000            CLOSE   =   65475
C000            CLRCHN  =   65484
C000            ZP      =   251
                                        ;
                                        ; Open a file to the printer with OPENPR,
                                        ; print a string with STRCPT, and
                                        ; close the channel with CLOSFL.
```

```
C000  20 10 C0  PRTSTR  JSR   OPENPR     ; open the printer
C003  20 27 C0          JSR   STRCPT     ; print the string
C006  A9 0D             LDA   #13        ; print RETURN to clear printer buffer
C008  20 D2 FF          JSR   CHROUT
C00B  A9 04             LDA   #4         ; file to close
C00D  4C 21 C0          JMP   CLOSFL     ; close file to printer; restore default device
                                         ; numbers and RTS
                                         ;
                                         ; OPEN the printer file as 4,4,0
C010  A9 04     OPENPR  LDA   #4         ; logical file 4
C012  A2 04             LDX   #4         ; device number; printer is usually device 4
                                         ; (sometimes 5)
C014  A0 00             LDY   #0         ; secondary address
C016  20 BA FF          JSR   SETLFS     ; set values
C019  20 C0 FF          JSR   OPEN       ; open a file to printer
C01C  A2 04             LDX   #4
C01E  4C C9 FF          JMP   CHKOUT     ; direct output to file 4 and RTS
                                         ;
                                         ; Closes the logical file specified in A and
                                         ; restores default devices.
C021  20 C3 FF  CLOSFL  JSR   CLOSE      ; close file in A
C024  4C CC FF          JMP   CLRCHN     ; clear all channels; restore default devices
                                         ; and RTS
                                         ;
                                         ; String printing routine
C027  A9 41     STRCPT  LDA   #<STRING   ; low byte of string address
C029  85 FB             STA   ZP         ; store it
C02B  A0 C0             LDY   #>STRING   ; high byte of string address
C02D  84 FC             STY   ZP+1       ; store it also
C02F  A0 00             LDY   #0         ; initialize index
C031  B1 FB     STRLOP  LDA   (ZP),Y     ; load each character from string
C033  F0 0B             BEQ   FINISH     ; zero byte marks end of string
C035  20 D2 FF          JSR   CHROUT     ; print character
C038  C8                INY              ; for next character
C039  D0 F6             BNE   STRLOP     ; if not more than 256 bytes, then get next
                                         ; character
C03B  E6 FC             INC   ZP+1       ; otherwise, increment high-byte address
                                         ; pointer to the string
C03D  4C 31 C0          JMP   STRLOP     ; and continue printing
C040  60        FINISH  RTS
                                         ;
C041  48 45 4C  STRING  .ASC  "HELLO"    ; string to print
C046  00                .BYTE 0          ; ending in zero byte
```

See also CLOSFL, OPENPR, PRTOUT.

Name

Print a string from a lookup table of addresses

Description

PTABAD is one of two routines presented in this book that print strings from a table (the other is **PTABCT**). With **PTABAD**, individual entries in a string table are given their own labels. A corresponding table of addresses for these labels is created. So, by indexing the address table, you can find the address of a particular string from the table.

As with **PTABCT**, each entry must end in a zero byte. In this case, the table itself can contain up to 127 separate entries. Strings within the table need not be of equal length since they are individually indexed.

Prototype

1. Enter the routine with .A holding the specified entry number. With ASL, multiply this number by 2.
2. Transfer the number of the entry requested (times 2) from .A to .X.
3. Store the address bytes, indexed by .X, of the chosen string in zero page.
4. Print the entry with **STRCPT**.

Explanation

The example program, with the aid of **PTABAD**, prints a word corresponding to a number in the range 0–9.

The program accepts only the number keys as input (see **CHRGTR**). The ASCII value of the number you specify is ANDed with 15, giving a number in the range 0–9.

After receiving a value, the program calls **PTABAD**, where the proper string is printed, and then waits for you to press another number key. To exit, press RUN/STOP–RESTORE.

Note: This method of accessing entries in a string table is faster than the method used in **PTABCT**, especially if there are a large number of entries. However, since each entry requires two additional addressing bytes (in ADRTAB), the multi-entry tables add to the length of the program. If you have a lot of short entries in your table, you may prefer to use **PTABCT** instead.

Routine

```
C000            GETIN    =    65508
C000            CHROUT   =    65490
C000            ZP       =    251
```

```
C000  20 E4 FF  WAIT     JSR    GETIN       ; get character code for key
C003  C9 30              CMP    #48         , compare with ASCII 0
C005  90 F9              BCC    WAIT        , too low, so get another keypress
C007  C9 3A              CMP    #58         ; compare with ASCII 9 plus 1
C009  B0 F5              BCS    WAIT        ; too high, so get another key
C00B  29 0F              AND    #15         ; to produce value 0-9
C00D  20 17 C0           JSR    PTABAD      , print corresponding string from table
C010  A9 0D              LDA    #13         , print RETURN
C012  20 D2 FF           JSR    CHROUT
C015  D0 E9              BNE    WAIT        ; look for another number
                                            ;
                                            , Enter with .A containing the entry number
                                            ; to print in the string table
C017  0A        PTABAD   ASL                ; multiply by 2 for offset into address table
C018  AA                 TAX                ; store number times 2 in .X
C019  BD 35 C0           LDA    ADRTAB,X    ; load low byte of address for number
C01C  85 [illegible]     STA    ZP          ; store in zero page
C01E  BD 36 C0           LDA    ADRTAB+1,X  ; also store high byte in zero page
C021  85 FC              STA    ZP+1
                                            ; Print out number string.
C023  A0 00     STRCPT   LDY    #0          ; initialize index
C025  B1 [illegible]  STRLOP   LDA    (ZP),Y      ; load each character from entry in string
                                            ; table
C027  F0 0B              BEQ    FINISH      ; if zero byte, you're finished
C029  20 D2 FF           JSR    CHROUT      ; print character
C02C  C8                 INY                ; next character
C02D  D0 F6              BNE    STRLOP      ; if .Y is not zero, get another character
C02F  E6 FC              INC    ZP+1        ; otherwise, increment high-byte address
                                            ; pointer to entry
C031  4C 25 C0           JMP    STRLOP      ; and continue printing
C034  60        FINISH   RTS
                                            ;
                                            ; ADRTAB contains two-byte addresses of
                                            ; each string entry
C035  49 C0 4E  ADRTAB   .WORD N0,N1,N2,N3,N4,N5,N6,N7,N8,N9
                                            ; string table
C049  5A 45 52  N0       .ASC   "ZERO"
C04D  00                 .BYTE0
C04E  4F 4E 45  N1       .ASC   "ONE"
C051  00                 .BYTE0
C052  54 57 4F  N2       .ASC   "TWO"
C055  00                 .BYTE0
C056  54 48 52  N3       .ASC   "THREE"
C05B  00                 .BYTE0
C05C  46 4F 55  N4       .ASC   "FOUR"
C060  00                 .BYTE0
C061  46 49 56  N5       .ASC   "FIVE"
C065  00                 .BYTE0
C066  53 49 58  N6       .ASC   "SIX"
C069  00                 .BYTE0
C06A  53 45 56  N7       .ASC   "SEVEN"
C06F  00                 .BYTE0
C070  45 49 47  N8       .ASC   "EIGHT"
C075  00                 .BYTE0
C076  4E 49 4E  N9       .ASC   "NINE"
C07A  00                 .BYTE0
```

See also PTABCT, STP128, STP64, STRCPT, STRLEN.

Name

Print a string from a table using a counting method

Description

This is the second of two routines that print string messages from a table (**PTABAD** is the other). **PTABCT** relies on the fact that individual strings in the table end with a zero byte. The table itself can contain up to 255 separate entries.

PTABCT, unlike many routines of this type, does not use an offset to address an individual table entry. Because of this, strings within the table need not be padded with spaces to insure they are equal in length.

Prototype

1. Enter with the address of the string table contained in .X (low byte) and .Y (high byte). Store the address in a zero page pointer.
2. Transfer the number of the entry requested from .A to .X.
3. If the specified entry number is zero, go to step 10 to print ZERO.
4. Read a byte from the STRING table.
5. If a byte is nonzero, branch to step 8.
6. Otherwise, decrement the entry counter in .X.
7. If the counter value has reached zero, go to step 9.
8. Update the zero-page pointer so that it points to the next byte and JMP to step 4.
9. Increment .Y so that it points to the first byte in the specified entry.
10. Print the chosen entry with **STRCPT**.

Explanation

This is a very flexible and useful routine. In a variety of programs, you'll require standard messages such as ARE YOU SURE?, PRESS ANY KEY, PLEASE WAIT, LOADING FILE, and so on. If you assign a number to each message, you can print any one of the messages by calling this routine.

In the example program, **PTABCT** is used to print a word corresponding to a number 0–9. Only the number keys (see **CHRGTR**) are acceptable input. The ASCII value of the number you choose is ANDed with 15, yielding a number 0–9.

Before JSRing to **PTABCT**, the address of the string table must be placed in the X and Y registers.

Basically, **PTABCT** operates by searching through the string table, character by character, until it comes upon a zero

byte, which indicates the end of another entry. At this point, the counter in .X is decremented. When the counter value reaches zero, the next entry is the chosen string.

After printing this string, the program waits for you to press another number key. To exit, press RUN/STOP–RESTORE.

Note: If your string table contains a considerable number of entries, the method used here—that is, counting through all the entries—may begin to slow down the program. In that case, use **PTABAD** where individual entries are addressed separately.

Routine

```
C000                GETIN    =    65508
C000                CHROUT   =    65490
C000                ZP       =    251

                                          ;
                                          ; Accept only keys 0–9 and print a string for
                                          ; the number from a table.
C000  20  E4  FF    WAIT     JSR  GETIN   ; get ASCII key
C003  C9  30                 CMP  #48     ; compare with ASCII 0
C005  90  F9                 BCC  WAIT    ; too low, so get another keypress
C007  C9  3A                 CMP  #58     ; compare with ASCII 9 + 1
C009  B0  F5                 BCS  WAIT    ; too high, so get another key
                                          ;
C00B  29  0F                 AND  #15     ; to produce value 0–9
C00D  A2  46                 LDX  #<STRTAB ; load string table address in .X and .Y
C00F  A0  C0                 LDY  #>STRTAB
C011  20  1C  C0             JSR  PTABCT  ; print string number corresponding to .A
C014  A9  0D                 LDA  #13     ; print RETURN
C016  20  D2  FF             JSR  CHROUT
C019  4C  00  C0             JMP  WAIT    ; get another number key
                                          ;
                                          ; Enter with entry number in .A, string table
                                          ; address in .X and .Y.
C01C  86  FB        PTABCT   STX  ZP      ; store low and high byte of string table
                                          ; address in zero page
C01E  84  FC                 STY  ZP+1
C020  A0  00                 LDY  #0      ; as an index in LOOP or STRLOP (if zero)
C022  AA                     TAX          ; use .X to hold input number
C023  F0  11                 BEQ  STRLOP  ; if zero, print it
C025  B1  FB        LOOP     LDA  (ZP),Y  ; load character from table
C027  D0  03                 BNE  INCZP   ; if not a zero byte
C029  CA                     DEX          ; if zero byte, decrement the counter
C02A  F0  09                 BEQ  STRCPT  ; counter is at zero, so print string from the
                                          ; table
C02C  E6  FB        INCZP    INC  ZP      ; to point to next character
C02E  D0  F5                 BNE  LOOP    ; if not on a page boundary, get next
                                          ; character
C030  E6  FC                 INC  ZP+1    ; otherwise, increment high byte of string
                                          ; address
C032  4C  25  C0             JMP  LOOP    ; and continue to look at characters
                                          ;
                                          ; Print out number string with STRCPT
C035  C8            STRCPT   INY          ; since string begins with next character
C036  B1  FB        STRLOP   LDA  (ZP),Y  ; load each character from an entry in string
                                          ; table
C038  F0  0B                 BEQ  FINISH  ; if zero byte, you're finished
C03A  20  D2  FF             JSR  CHROUT  ; print character
```

```
C03D  C8                        INY                 ; next character
C03E  D0 F6                     BNE    STRLOP       ; if .Y is not zero, get another character
C040  E6 FC                     INC    ZP+1         ; otherwise, increment high-byte address
                                                    ; pointer to entry
C042  4C 36 C0                  JMP    STRLOP       ; and continue printing
C045  60              FINISH    RTS
                                                    ; string table
C046  5A 45 52        STRTAB    .ASC   "ZERO"
C04A  00                        .BYTE0
C04B  4F 4E 45                  .ASC   "ONE"
C04E  00                        .BYTE0
C04F  54 57 4F                  .ASC   "TWO"
C052  00                        .BYTE0
C053  54 48 52                  .ASC   "THREE"
C058  00                        .BYTE0
C059  46 4F 55                  .ASC   "FOUR"
C05D  00                        .BYTE0
C05E  46 49 56                  .ASC   "FIVE"
C062  00                        .BYTE0
C063  53 49 58                  .ASC   "SIX"
C066  00                        .BYTE0
C067  53 45 56                  .ASC   "SEVEN"
C06C  00                        .BYTE0
C06D  45 49 47                  .ASC   "EIGHT"
C072  00                        .BYTE0
C073  4E 49 4E                  .ASC   "NINE"
C077  00                        .BYTE0
```

See also PTABAD, STP128, STP64, STRCPT, STRLEN.

Name

Set up a raster interrupt

Description

This routine seemingly performs magic. Instead of one screen, suddenly there are two half-screens, each with its own background color and eight sprites. Running the sample BASIC program gives you a total of 16 independent sprites (each limited to one half of the screen or the other) which can be displayed at the same time.

Prototype

This is a two-part routine. In the first part, **RAS64**:

1. Disable all CIA #1 IRQ interrupt sources.
2. Redirect the IRQ interrupt vector at 788 to the main raster interrupt routine (MAIN).
3. Clear the ninth bit of the raster compare register (bit 7 of location 53265).
4. Enable the raster compare IRQ interrupt.
5. Create two sets of shadow registers for the VIC-II chip registers (53248–53294) by copying them twice into free memory.
6. Then RTS.

In MAIN:

1. Prevent other interrupts from occurring by clearing the interrupt condition.
2. Determine where the last raster line was drawn by reading the raster compare register at 53266.
3. If it was less than 147, store a 147 into the raster register so the next raster interrupt occurs at this line (the middle of the screen). Otherwise, store a one in this register so the raster interrupt occurs at the top of the screen.
4. Allow the current raster line to finish drawing and then copy the appropriate set of shadow registers into the VIC-II chip (representing either the top or bottom of the screen).
5. Check the interrupt control register (CIAICR) for a Timer A interrupt. If one has occurred, execute the normal IRQ service routine. Otherwise, restore the stack and RTI.

Explanation

On the 64, the normal hardware interrupt happens 60 times a second (50 times per second on European 64s). One of the CIA chips is given the responsibility of counting down and

triggering an interrupt after a certain period of time has elapsed. The hardware interrupt is a maskable interrupt request (IRQ), not a nonmaskable interrupt (NMI). *Maskable* means it can be turned off.

The hardware interrupt is important because it causes the CPU (the brains of the 64) to pause what it's doing and service the interrupt. During the service routine, the cursor blinks, the keyboard is checked for keypresses, and the jiffy clock is updated.

The ML program below first turns off the normal interrupt. It will no longer be triggered by the CIA clock. Instead, we turn on a different interrupt, one caused by the position of the raster on the screen. North American TVs and monitors normally use 525 raster lines per screen, but the 64 draws only half this many, so there are effectively 262.5 lines per screen. Of these, 200 make up the text screen and the additional lines form the top and bottom borders. The raster lines of the visible screen are numbered 50–250. The halfway point on the screen is raster line 150.

The IRQVEC at 788 normally points to the interrupt service routine (which reads the keyboard and handles the other housekeeping chores). The first thing we do after disabling the interrupt is change the vector to point to our routine. Next, the raster interrupt is turned on and we make two copies of the VIC chip registers, one at $C100 (49408) and the other 47 bytes higher.

Now interrupts are triggered when the raster beam reaches a certain line on the screen. When line 147 appears, suddenly an interrupt occurs. The register RASTER does two things. If you read it, it tells you which line is being drawn. If you write to it, you set the value for a raster interrupt. If the raster is in the middle of the screen, we want to enable a new raster interrupt to happen at line 1. If the raster is at line 1, we change the interrupt to happen at line 147. After each interrupt, the main routine copies one of the two shadows of the VIC chip to the VIC chip.

Since there are two complete copies of the VIC chip, you can treat the two halves of the screen as two separate screens. One could be in multicolor hi-res mode while the other is displaying normal text. You can give each half separate border and background colors. Each halfscreen has its own eight sprites, with which you can do what you please.

After assembling and SYSing to the **RAS64** program, type in and run the following short BASIC program to see the effects of the raster interrupt:

```
10 PRINT CHR$(147):POKE 49408+33,0:POKE 49455+33,0:REM
   BACKGROUND BLACK
15 FOR A=832 TO 896:POKE A,255:NEXT:REM DEFINE BLOCK SPRITE
20 FOR A=2040 TO 2047:POKE A,13:NEXT:REM SET SPRITE POINT-
   ERS TO BLOCK SPRITE
30 POKE 49408+21,255:POKE 49455+21,255:REM ENABLE SPRITES
   (TOP/BOTTOM)
39 REM HORIZONTAL POSITION (TOP/BOTTOM)
40 FOR A=49408 TO 49422 STEP 2:POKE A,B*25+50:POKE
   A+47,B*25+50:B=B+1:NEXT
49 REM VERTICAL POSITION (TOP/BOTTOM)
50 FOR A=49409 TO 49423 STEP 2:POKE A,100:POKE A+47,200:NEXT
```

Routine

```
C000                VIC      =    53248        ; start of VIC chip registers
C000                NEWVIC   =    49408        ; shadow registers for VIC chip
C000                CIAICR   =    56333        ; interrupt control register
C000                SCROLY   =    53265        ; scrolling/control register (bit 7 is high bit of
                                               ; raster)
C000                IRQMSK   =    53274        ; IRQ mask register
C000                VICIRQ   =    53273        ; VIC interrupt flag register
C000                RASTER   =    53266        ; read/write raster compare register
C000                IRQVEC   =    788          ; IRQ interrupt vector
C000                IRQNOR   =    59953        ; normal IRQ handler routine
C000                IRQEND   =    65212        ; end of IRQ interrupt handler (clean stack
                                               ; and RTI)
                                               ;
C000  A9 7F         RAS64    LDA  #$7F
C002  8D 0D DC               STA  CIAICR       ; turn off CIA #1 interrupts
C005  A9 28                  LDA  #<MAIN       ; redirect IRQ interrupt vector to main, low
                                               ; byte first
C007  8D 14 03               STA  IRQVEC
C00A  A9 C0                  LDA  #>MAIN       ; then high byte
C00C  8D 15 03               STA  IRQVEC+1
C00F  A9 1B                  LDA  #%00011011
C011  8D 11 D0               STA  SCROLY       ; clear high bit of raster compare register
C014  A9 01                  LDA  #1
C016  8D 1A D0               STA  IRQMSK       ; enable raster interrupts
C019  A0 2E                  LDY  #46          ; index for COPY
C01B  [redacted] 00 D0      COPY     LDA  VIC,Y        ; copy 47 VIC registers as two sets of
                                               ; shadow registers
C01E  99 00 C1               STA  NEWVIC,Y     ; initialize shadow registers for top of
                                               ; screen (set 1)
C021  99 2F C1               STA  NEWVIC+47,Y
                                               ; initialize shadow registers for bottom of
                                               ; screen (set 2)
C024  88                     DEY               ; next lower VIC register
C025  10 F4                  BPL  COPY         ; are all copied?
C027  60                     RTS
                                               ;
                                               ; Main raster interrupt routine follows.
C028  A9 01         MAIN     LDA  #1
```

```
C02A  8D 19 D0          STA  VICIRQ     ; prevent normal raster—clear interrupt
                                        ; condition
C02D  A2 93             LDX  #147       ; raster line in the middle of screen
C02F  A0 2E             LDY  #46        ; index for VIC registers to copy for top of
                                        ; the screen (set 1)
C031  AD 12 D0          LDA  RASTER     ; get the current raster line number
C034  C9 93             CMP  #147       ; determine if it's on the top half of screen
C036  90 04             BCC  TOP        ; if so, skip to TOP
C038  A2 01             LDX  #1         ; raster line for top of screen
C03A  A0 5D             LDY  #93        ; index for set 2 registers (bottom of screen
                                        ; registers)
C03C  8A        TOP     TXA             ; raster line becomes 1 (if now on bottom)
                                        ; or 147 (if now on top)
C03D  48                PHA             ; save it temporarily
C03E  A2 03             LDX  #3         ; wait for current raster line to finish
                                        ; drawing
C040  CA        DELAY   DEX
C041  D0 FD             BNE  DELAY
C043  EA                NOP             ; slight adjustment to DELAY
C044  A2 2E             LDX  #46        ; index for COPYBK
C046  B9 00 C1  COPYBK  LDA  NEWVIC,Y   ; copy from set 1 or 2 VIC shadow registers
C049  9D 00 D0          STA  VIC,X      ; to VIC registers
C04C  88                DEY
C04D  CA                DEX
C04E  10 F6             BPL  COPYBK     ; copy 47 values
C050  68                PLA             ; get new raster line (1 or 147)
C051  8D 12 D0          STA  RASTER     ; set raster for next interrupt
C054  AD 0D DC          LDA  CIAICR     ; bit 1 set if IRQ interrupt is needed
C057  4A                LSR
C058  90 03             BCC  NOIRQ      ; bit is clear so no IRQ interrupts
C05A  4C 31 EA          JMP  IRQNOR     ; otherwise, call normal IRQ interrupt
                                        ; routine
C05D  4C BC FE  NOIRQ   JMP  IRQEND     ; clean up stack and RTI
```

See also IRQINT, NMIINT, RAS128.

Name

Set up a raster interrupt

Description

This is the 128 version of RAS64. It splits the screen in two and provides two shadows of the VIC chip, which can be set to any of the video modes (hi res, multicolor hi res, or text). Each half has its own eight sprites as well.

Prototype

This is a two-part routine. In the first part, **RAS128**:

1. Disable all IRQ interrupt sources.
2. Redirect the IRQ interrupt vector at 788 to the main raster interrupt routine (MAIN).
3. Clear the ninth bit of the raster compare register (bit 7 of location 53265).
4. Create two sets of shadow registers for the VIC-II chip registers (53248–53294) by copying them twice into free memory.
5. Reenable IRQ interrupt sources and then RTS.

In MAIN:

1. Clear decimal mode as required by the normal IRQ interrupt handler.
2. Prevent normal raster interrupts from occurring by clearing the interrupt condition.
3. Determine where the last raster line was drawn by reading the raster compare register at 53266.
4. If it was less than 147, store a 147 into the raster register so the next raster interrupt occurs at this line (the middle of the screen). Otherwise, store a one in this register so the raster interrupt occurs at the top of the screen.
5. Allow the current raster line to finish drawing and then copy the appropriate set of shadow registers into the VIC-II chip (for either the top or bottom of the screen).
6. Check a flag to see if the cursor needs blinking (every other time through the routine). If so, execute the normal IRQ interrupt handler routine (except for the any raster-related routines). Otherwise, leave through the common interrupt exit point at 65331.

Explanation

For a more detailed explanation of what interrupts are, see the **RAS64** routine. Much of this program is very similar to **RAS64**. It assembles to $0C00 on the 128, and the shadows of the VIC chip are at 3328 ($0D00).

After assembling and SYSing to the ML raster interrupt routine, run this short BASIC program to see the effects of the raster split:

```
10 SCNCLR:POKE 2564,0:REM TURN OFF NORMAL SPRITE
   ROUTINES
15 FOR A=3584 TO 3647:POKE A,255:NEXT:REM DEFINE BLOCK
   SPRITE
20 FOR A=2040 TO 2047:POKE A,56:NEXT:REM SET POINTERS TO
   BLOCK SPRITE DATA
30 POKE 3328+21,255:POKE 3375+21,255: REM ENABLE SPRITES FOR
   TOP/BOTTOM
39 REM HORIZONTAL POSITIONS (TOP/BOTTOM)
40 FOR A=3328 TO 3342 STEP 2:POKE A,B*25+50:POKE
   A+47,B*25+50:B=B+1:NEXT
49 REM VERTICAL POSITIONS (TOP/BOTTOM)
50 FOR A=3329 TO 3343 STEP 2:POKE A,100:POKE A+47,200:NEXT
```

Routine

```
0C00                VIC       =     53248        ; start of VIC chip
0C00                NEWVIC    =     3328         ; shadow registers for VIC chip
0C00                VICIRQ    =     53273        ; VIC interrupt flag register
0C00                RASTER    =     53266        ; read/write raster compare register
0C00                IRQVEC    =     788          ; IRQ interrupt vector
0C00                IRQTXT    =     49636        ; text-mode portion of IRQ editor routine
0C00                IRQNRP    =     64107        ; entry point to IRQ handler just beyond
                                                 ; raster handler
0C00                CRTI      =     65331        ; interrupt exit routine (clean stack and RTI)
0C00                ZP        =     251
                                                 ;
0C00  78            RAS128    SEI                ; disable all IRQ interrupts
0C01  A9 1B                   LDA   #<MAIN       ; redirect IRQ interrupt vector to main, low
                                                 ; byte first
0C03  8D 14 03                STA   IRQVEC
0C06  A9 0C                   LDA   #>MAIN       ; then high byte
0C08  8D 15 03                STA   IRQVEC+1
0C0B  A0 2E                   LDY   #46          ; index for COPY
0C0D  B9 00 D0      COPY      LDA   VIC,Y        ; copy 47 VIC registers as two sets of
                                                 ; shadow registers
0C10  99 00 0D                STA   NEWVIC,Y     ; initialize shadow registers for top of
                                                 ; screen (set 1)
0C13  99 2F 0D                STA   NEWVIC+47,Y
                                                 ; initialize shadow registers for bottom of
                                                 ; screen (set 2)
0C16  88                      DEY                ; next lower VIC register
0C17  10 F4                   BPL   COPY         ; are all copied?
0C19  58                      CLI                ; reenable IRQ interrupts
0C1A  60                      RTS
```

```
                                        ;
                                        ; Main raster interrupt routine follows.
0C1B  D8          MAIN    CLD           ; clear decimal mode (required by normal
                                        ; IRQ handler)
0C1C  A9 01               LDA  #1
0C1E  8D 19 D0            STA  VICIRQ   ; prevent normal raster—clear interrupt
                                        ; condition
0C21  A2 93               LDX  #147     ; raster line in the middle of screen
0C23  A0 2E               LDY  #46      ; index for VIC registers to copy for top of
                                        ; the screen (set 1)
0C25  AD 12 D0            LDA  RASTER   ; get the current raster line number
0C28  C9 93               CMP  #147     ; determine if it's on the top half of screen
0C2A  90 04               BCC  TOP      ; if so, skip to TOP
0C2C  A2 01               LDX  #1       ; raster line for top of screen
0C2E  A0 5D               LDY  #93      ; index for set 2 shadow registers (bottom
                                        ; of screen registers)
0C30  8A          TOP     TXA           ; raster line becomes 1 (if now on bottom)
                                        ; or 147 (if now on top)
0C31  48                  PHA           ; save it temporarily
0C32  A2 0A               LDX  #10      ; wait for current raster line to finish
                                        ; drawing
0C34  CA          DELAY   DEX
0C35  D0 FD               BNE  DELAY
0C37  A2 2E               LDX  #46      ; index for COPYBK
0C39  B9 00 0D    COPYBK  LDA  NEWVIC,Y ; copy from set 1 or 2 VIC shadow registers
0C3C  9D 00 D0            STA  VIC,X    ; to VIC registers
0C3F  88                  DEY
0C40  CA                  DEX
0C41  10 F6               BPL  COPYBK   ; copy 47 values
0C43  68                  PLA           ; get new raster line (1 or 147)
0C44  8D 12 D0            STA  RASTER   ; set raster for next interrupt
0C47  A5 FB               LDA  ZP       ; flag for cursor
0C49  49 80               EOR  #128     ; flip it to positive or negative
0C4B  85 FB               STA  ZP       ; save result for next pass
0C4D  10 07               BPL  NOCURS   ; only go to the cursor routine half the time
0C4F  38                  SEC           ; required by following routine
0C50  20 E4 C1            JSR  IRQTXT   ; go to text-mode portion of IRQ editor
                                        ; routine, skipping raster
0C53  4C 6B FA            JMP  IRQNRP   ; continue beyond normal raster routine
0C56  4C 33 FF    NOCURS  JMP  CRTI     ; clean the stack and RTI (common
                                        ; interrupt exit point)
```

See also IRQINT, NMIINT, RAS64.

Name

Generate a random two-byte integer value using SID voice 3

Description

RNDBYT returns a one-byte random integer using voice 3 of the SID chip. **RD2BYT** also relies on voice 3 to generate a random integer value. This time, two separate bytes are returned. One represents the high byte of the number; the other, the low byte. A random two-byte integer value in the range 0–65535 is produced.

Prototype

In an initialization routine (RDINIT):

1. Set voice 3 to a high frequency
2. Select the noise waveform.
3. Turn off the SID chip volume and disconnect the output of voice 3.

In **RD2BYT** itself:

1. Load a random byte value from voice 3's random number generator (RANDOM) into .X.
2. Cause a delay of two jiffies.
3. Load a second value from RANDOM into .A.

Explanation

In the example program, a random two-byte integer is generated by **RD2BYT** and printed on the screen.

The setup for **RD2BYT** is the same as in **RNDBYT**. Voice 3's random number generator is first initialized by JSRing to RDINIT. For a full explanation of how the random number generator is accessed, refer to **RNDBYT**.

After the random number generator has been initialized, two individual random byte values are taken from RANDOM (54299) within **RD2BYT**. One is returned in the X register, and the other in the accumulator. It really doesn't matter which is which.

Notice that between taking these two bytes, a delay of two jiffies (a total of 2/60 second) is carried out. This insures that the current waveform has had time to change before the next byte is taken. If not for this delay, the two bytes would be very close in value, and we'd lose our randomness.

Routine

```
C000                GETIN    =    65508
C000                LINPRT   =    48589         ; LINPRT = 36402 on the 128
C000                FREHI3   =    54287         ; voice 3 frequency control (high byte)
C000                VCREG3   =    54290         ; voice 3 control register
C000                SIGVOL   =    54296         ; volume and filter select register
C000                RANDOM   =    54299         ; oscillator 3/ random number generator
C000                JIFFY    =    162           ; jiffy clock (jiffies)
                                                ;
                                                ; Generate a random integer (0-65535) from
                                                ; SID chip voice 3.
C000  20 09 C0  MAIN     JSR  RDINIT            ; initialize SID voice 3 for random numbers
C003  20 17 C0  LOOP     JSR  RD2BYT            ; get a random two-byte integer
C006  4C CD BD  NUMOUT   JMP  LINPRT            ; two random bytes are in .A and .X
                                                ; So print the resulting two-byte integer (see
                                                ; NUMOUT).
                                                ;
                                                ; Routine to initialize SID voice 3 for random
                                                ; numbers.
C009  A9 FF     RDINIT   LDA  #$FF              ; set voice 3 frequency (high byte) to
                                                ; maximum
C00B  8D 0F D4           STA  FREHI3
C00E  A9 80              LDA  #%10000000
C010  8D 12 D4           STA  VCREG3            ; select noise waveform and start release
C013  8D 18 D4           STA  SIGVOL            ; turn off volume and disconnect output of
                                                ; voice 3
C016  60                 RTS
                                                ;
                                                ; RD2BYT returns a two-byte integer in .X
                                                ; and .A.
C017  AE 1B D4  RD2BYT   LDX  RANDOM            ; get single-byte random number
C01A  A5 A2              LDA  JIFFY             ; pseudorandom delay
C01C  69 02              ADC  #2
C01E  C5 A2     DELAY    CMP  JIFFY             ; wait till jiffy clock reads the original
                                                ; value plus 2
C020  D0 FC              BNE  DELAY             ; otherwise, wait
C022  AD 1B D4           LDA  RANDOM            ; get a second random byte
C025  60                 RTS
```

See also RDBYRG, RND1VL, RNDBYT.

Name

Open a disk channel, read a sector, copy the disk buffer to memory

Description

This is a fairly low-level routine for reading a given disk sector into a buffer inside the drive. The 256 numbers in the buffer are then read byte by byte into the computer's memory.

Prototype

1. Open the command channel (15,8,15).
2. Open a disk buffer (equivalent to BASIC OPEN 1,8,3,"#").
3. Read the buffer by sending read sector command to channel 15.
4. Perform a Kernal CHKIN to logical file 1.
5. Read the 256 bytes into memory with CHRIN.
6. Close all channels and exit.

Explanation

The example program reads track 18, sector 1 (the first of the directory sectors), into memory. There are several discrete sections of the routine.

First, the disk command channel must be opened ($C044–$C05A) using secondary address 15. Next, an internal disk buffer is allocated, with the equivalent of OPEN 1,8,3,"#", at $C05B–$C075. The secondary address, 3 in this case, is important. It must be used in commands to the drive.

The string *U1,3,0,18,1* sends five pieces of information to channel 15 ($C006–$C01D). *U1* is the sector-read command to the disk drive. The 3 corresponds to the secondary address of the buffer (the 3 in OPEN 1,8,3). The *0* is the drive number (if you have an MSD dual drive, you could use 1). The *18* and *1* are the track and sector numbers, respectively, for the block to be read.

When the 1541 or 1571 receives the U1 command, it copies the given disk sector into memory inside the disk drive. All that remains is to read the data into the computer's memory. At this point, we CHKIN with a 1 (the 1 in OPEN 1,8,3) to specify logical file 1 as the channel to be read and then loop 256 times with CHRIN to read the bytes and store them.

Finally, logical files 1 and 15 are closed and the routine is done.

RDBUFF

Routine

```
C000                SETLFS   =    $FFBA
C000                SETNAM   =    $FFBD
C000                OPEN     =    $FFC0
C000                CHKOUT   =    $FFC9
C000                CHKIN    =    $FFC6
C000                CHROUT   =    $FFD2
C000                CHRIN    =    $FFCF
C000                CLOSE    =    $FFC3
C000                CLRCHN   =    $FFCC

C000  20 44 C0  RDBUFF   JSR  OPEN15
C003  20 5B C0           JSR  OPNBUF
C006  A2 0F              LDX  #15
C008  20 C9 FF           JSR  CHKOUT      ; ready to send to logical file 15
C00B  90 03              BCC  OUTOK       ; carry clear if no error
C00D  4C 76 C0           JMP  ERROR       ; else print error message
C010  A0 00     OUTOK    LDY  #0          ; initialize index
C012  B9 6B C0  LOOP1    LDA  BLKRD,Y     ; send the command
C015  F0 07              BEQ  DONEBR      ; if 0 we're done setting up the block read
                                          ; command
C017  20 D2 FF           JSR  CHROUT      ; else send the next character
C01A  C8                 INY              ; increment index
C01B  4C 12 C0           JMP  LOOP1       ; and go back for another
C01E  20 CC FF  DONEBR   JSR  CLRCHN      ; back to normal I/O
                                          ;
C021  A2 01              LDX  #1          ; open logical file 1
C023  20 C6 FF           JSR  CHKIN       ; for input
C026  90 03              BCC  INPOK       ; carry clear if no error
C028  4C 76 C0           JMP  ERROR       ; otherwise, print error message
C02B  A0 00     INPOK    LDY  #0          ; start counter at zero
C02D  20 CF FF  GETEM    JSR  CHRIN       ; get a character from the buffer
C030  99 B2 C0           STA  MEMORY,Y    ; store (indexed) to memory
C033  C8                 INY              ; count 0-255
C034  D0 F7              BNE  GETEM       ; wraps around to 0 at end
C036  A9 01     FINIS    LDA  #1
C038  20 C3 FF           JSR  CLOSE       ; close logical file 1
C03B  A9 0F              LDA  #15
C03D  20 C3 FF           JSR  CLOSE       ; and the command channel
C040  20 CC FF           JSR  CLRCHN      ; and clear the channels
C043  60                 RTS
                                          ; Subroutines
C044  A9 0F     OPEN15   LDA  #15         ; file number
C046  A2 08              LDX  #8          ; device number for disk drive
C048  A0 0F              LDY  #15         ; secondary address for command channel
C04A  20 BA FF           JSR  SETLFS      ; 15,8,15 is set to be opened
C04D  A9 00              LDA  #0          ; length of name is zero
C04F  20 BD FF           JSR  SETNAM
C052  20 C0 FF           JSR  OPEN        ; open logical file
C055  90 03              BCC  OK15        ; check for error
C057  4C 76 C0           JMP  ERROR       ; print message if there's a problem
C05A  60        OK15     RTS

                                          ;
                                          ; OPNBUF opens a disk buffer for reading.
C05B  A9 01     OPNBUF   LDA  #1          ; logical file number
C05D  A2 08              LDX  #8          ; disk drive
C05F  A0 03              LDY  #3          ; secondary address
C061  20 BA FF           JSR  SETLFS
C064  A9 01              LDA  #1          ; one character
C066  A2 8A              LDX  #<BUFNAM    ; the # specifies a drive buffer
C068  A0 C0              LDY  #>BUFNAM
C06A  20 BD FF           JSR  SETNAM      ; set up the name
C06D  20 C0 FF           JSR  OPEN        ; now it's ready
```

```
C070  90 03              BCC   OKBUF      ; to OKBUF if no error
C072  4C 76  C0          JMP   ERROR      ; jump to ERROR if there is
C075  60         OKBUF   RTS
                                          ;
                                          ; ERROR prints a message if a disk error
                                          ; occurs
C076  20 CC  FF  ERROR   JSR   CLRCHN     ; close down and clear channels
C079  A0 00              LDY   #0         ; initialize index
C07B  B9 98  C0  MORE    LDA   ERRMSG,Y
C07E  F0 07              BEQ   MSGEND     ; message ends with zero byte
C080  20 D2  FF          JSR   CHROUT     ; print the character
C083  C8                 INY              ; increment the index
C084  4C 7B  C0          JMP   MORE       ; and go back
C087  4C 36  C0  MSGEND  JMP   FINIS      ; finish closing files
                                          ;
                                          ; Variables
C08A  23         BUFNAM  .ASC  "#"
C08B  55 31  2C  BLKRD   .ASC  "U1,3,0,18,1"
                                          ; U1 is block read
                                          ; 3 is secondary address,
                                          ; 0 means drive zero
                                          ; track 18, sector 1
C096  0D 00              .BYTE 13,0
C098  41 20  44  ERRMSG  .ASC  "A DISK ERROR HAS OCCURRED"
C0B1  00                 .BYTE 0
C0B2             MEMORY  =     *
C1B2                     *=    * + 256
                                          ; Reserve 256 bytes for data from sector read
                                          ; from disk.
```

See also WRBUFF.

Name

Generate a random one-byte integer in a range

Description

A routine for generating a random one-byte value in the range 0–255 has been provided (**RNDBYT**). Frequently, though, a random value must be limited to a particular range.

For example, in a game, you might wish to position a sprite or a character randomly within a certain range of rows or columns. Or in an educational program, you might want to pick two numbers in the range 11–20 (for adding or multiplying, say).

Prototype

In an initialization routine (**RDINIT**):

1. Set voice 3 to a high frequency.
2. Select the noise waveform.
3. Turn off the SID chip volume and disconnect the output of voice 3.

In **RDBYRG** itself:

1. Load a random byte value from voice 3's random number generator (RANDOM) into .A.
2. Determine whether this value lies within the acceptable range (here, delimited by LOWLIM and UPPLIM−1).
3. If not, branch to step 1 for another value.
4. Otherwise, return this suitable integer in .A.

Explanation

Ten random integers in the range 30–45 are generated by the example program and are printed to the screen.

In **RNDBYT**, a random byte value is generated by using voice 3 of the SID chip. A similar approach is taken here except that we limit the range of the number.

Again, a two-part routine is required. The first part (RDINIT) is responsible for initializing the random number generator of voice 3 (RANDOM). This is done by selecting the noise waveform and setting it to its maximum frequency. For a more detailed description of how this is accomplished, refer to **RNDBYT**.

Once the random number generator has been initialized at the outset of your main program, random values can be taken from RANDOM within **RDBYRG**. If a value falls within the range set by LOWLIM and UPPLIM (minus 1), it's accepted

and returned in the accumulator. Otherwise, another random number is fetched.

In using **RDBYRG** within your own programs, be sure to define the range delimiters before the routine is entered. For instance, to generate a random integer in the range 1–10, change LOWLIM to 1, and UPPLIM to 11 (1 plus the actual upper limit).

Routine

```
C000                CHROUT  =     65490
C000                LINPRT  =     48589       ; LINPRT = 36402 on the 128
C000                FREHI3  =     54287       ; voice 3 frequency control (high byte)
C000                VCREG3  =     54290       ; voice 3 control register
C000                SIGVOL  =     54296       ; volume and filter select register
C000                RANDOM  =     54299       ; oscillator 3/random number generator
                                              ;
                                              ; Generate ten random byte values using SID
                                              ; chip voice 3 in a range (30-45)
                                              ; and print them.
C000  20 1C C0  MAIN     JSR   RDINIT         ; initialize SID voice 3 for random numbers
C003  A9 0A              LDA   #10            ; initialize counter for ten random numbers
C005  8D 38 C0           STA   TEMCNT         ; save counter
C008  20 2A C0  LOOP     JSR   RDBYRG         ; get random byte in a range
C00B  AA                 TAX                  ; move value to .X
C00C  A9 00              LDA   #0             ; zero for high byte (in .A)
C00E  20 CD BD           JSR   LINPRT         ; print the number
C011  A9 0D              LDA   #13            ; print a RETURN
C013  20 D2 FF           JSR   CHROUT
C016  CE 38 C0           DEC   TEMCNT         ; decrement counter
C019  D0 ED              BNE   LOOP           ; if not ten values, then loop
C01B  60                 RTS
                                              ;
                                              ; Initialize SID voice 3 for random numbers.
C01C  A9 FF     RDINIT   LDA   #$FF           ; set voice 3 frequency (high byte) to
                                              ; maximum
C01E  8D 0F D4           STA   FREHI3
C021  A9 80              LDA   #%10000000
C023  8D 12 D4           STA   VCREG3         ; select noise waveform and start release
C026  8D 18 D4           STA   SIGVOL         ; turn off volume and disconnect output of
                                              ; voice 3
C029  60                 RTS
                                              ;
                                              ; Returns a random byte in a range.
C02A  AD 1B D4  RDBYRG   LDA   RANDOM         ; get single-byte random number
C02D  CD 39 C0           CMP   LOWLIM         ; lower limit of range
C030  90 F8              BCC   RDBYRG
C032  CD 3A C0           CMP   UPPLIM         ; upper limit of range
C035  B0 F3              BCS   RDBYRG
C037  60                 RTS
                                              ;
C038  00        TEMCNT   .BYTE 0              ; temporary storage for counter
C039  1E        LOWLIM   .BYTE 30             ; lowest possible number
C03A  2E        UPPLIM   .BYTE 46             ; highest possible number plus 1
```

See also RD2BYT, RND1VL, RNDBYT.

Name

Check the I/O status by using the Kernal READST routine

Description

Although some Kernal routines have their own ways of flagging errors, the READST routine is a general routine that returns an error flag if something has gone wrong with an input or output operation. It's most often used to check the status of the disk drive.

Prototype

1. JSR to the READST routine.
2. If the equal flag is set, everything's okay. Otherwise, an error has occurred.

Explanation

The following program deliberately causes a disk error by trying to open a file with no name. Then it calls READST to see if anything's wrong. If an error has occurred, the letter *A* prints to the screen. Otherwise, the program ends.

Note that **RDSTAT** is similar to **CHK144**. Both return a zero as long as the situation is in hand. When an error occurs, the result is a nonzero value.

Routine

```
C000                    SETLFS   =     $FFBA
C000                    SETNAM   =     $FFBD
C000                    OPEN     =     $FFC0
C000                    READST   =     $FFB7
C000                    CHROUT   =     $FFD2
C000                    CHKOUT   =     $FFC9
C000                    CLRCHN   =     $FFCC
C000                    CLOSE    =     $FFC3

C000  A9 02                      LDA   #2
C002  A2 08                      LDX   #8
C004  A0 02                      LDY   #2
C006  20 BA FF                   JSR   SETLFS    ; set file parameters
C009  A9 00                      LDA   #0
C00B  20 BD FF                   JSR   SETNAM    ; no name
C00E  20 C0 FF                   JSR   OPEN      ; open it
C011  A2 02                      LDX   #2
C013  20 C9 FF                   JSR   CHKOUT    ; get ready to print
C016  20 B7 FF  RDSTAT           JSR   READST    ; check the status
C019  F0 08                      BEQ   FINIS     ; if equal to zero, OK
C01B  20 CC FF                   JSR   CLRCHN    ; clear channels before printing
C01E  A9 41                      LDA   #65
C020  20 D2 FF                   JSR   CHROUT    ; print a letter A
C023  20 CC FF  FINIS            JSR   CLRCHN    ; clear all channels
C026  A9 02                      LDA   #2
C028  20 C3 FF                   JSR   CLOSE     ; and close file 2
C02B  60                         RTS
```

See also CHK144, DERRCK.

Name

Read and write to the 80-column video chip

Description

These two short routines, **RE80CO** and **WR80CO**, read values from or write values to the VDC chip's internal registers.

Prototype

1. Enter either routine with .X holding the register number.
2. Store it into the first gateway byte $D600.
3. Wait for bit 7 of the gateway byte to go high.
4. LDA from or STA to the second gateway byte.

Explanation

The 128's VDC chip has 36 internal registers and 16K of private RAM. But the only way to access the chip is through locations 54784 and 54785 ($D600 and $D601). You must store into the first gateway byte the number of the register you wish to get to. The second gateway byte can then be PEEKed or POKEd to read or write the value from the register whose number you put in the first byte.

The example program POKEs the values 1–5 to the screen. You should see the letters *A–E* appear on your monitor (if it is set for an 80-column display). First, the internal address of the screen is read from VDC registers 12–13. This value is stored into the memory access registers (18–19). Once the memory access registers know the place to read or write, the values from MESSAGE are sent to the read/write register (31).

Routine

```
0C00              SCRHIR   =    12
0C00              SCRLOR   =    13          ; high and low bytes of the register for screen
                                            ; memory
0C00              MEMHIR   =    18
0C00              MEMLOR   =    19          ; high and low bytes for getting to memory
0C00              GATE     =    31          ; the read/write register
0C00              VDCADR   =    $D600
0C00              VDCDAT   =    $D601
0C00              START    =    *
0C00  A2 0C                LDX  #SCRHIR     ; find the high byte of screen memory from
                                            ; register 12 ($0C)
0C02  20 24 0C             JSR  RE80CO      ; read it from 12
0C05  A2 12                LDX  #MEMHIR     ; now send it to memory write (high) register
0C07  20 30 0C             JSR  WR80CO      ; write .A to the register in .X
0C0A  A2 0D                LDX  #SCRLOR     ; now do the low byte
0C0C  20 24 0C             JSR  RE80CO      ; read it
0C0F  A2 13                LDX  #MEMLOR     ; low byte of memory-write
0C11  20 30 0C             JSR  WR80CO      ; and write it
                                            ;
                                            ; Now the internal registers are set up.
```

```
0C14  A0 00               LDY   #0           ; the index
0C16  A2 1F      MORE     LDX   #GATE        ; set up the gateway byte
0C18  B9 3C 0C            LDA   MESSAGE,Y    ; get a screen code
0C1B  F0 06               BEQ   ALLDONE      ; if zero, we're finished
0C1D  20 30 0C            JSR   WR80CO       ; write to register 31
0C20  C8                  INY
0C21  D0 F3               BNE   MORE         ; keep looping
0C23  60         ALLDONE  RTS
                                             ;
                                             ; Enter RE80CO with the internal register
                                             ; in .X.
                                             ;
0C24  8E 00 D6   RE80CO   STX   VDCADR       ; tell the 8563 we want to access a register
0C27  AE 00 D6   LOOP1    LDX   VDCADR       ; check the door
0C2A  10 FB               BPL   LOOP1        ; if bit 7 is clear, the door is locked
0C2C  AD 01 D6            LDA   VDCDAT       ; else, get the byte from the internal
                                             ; register
0C2F  60                  RTS
                                             ; Exit with the value in .A.
                                             ;
                                             ; Enter WR80CO with the register in .X, the
                                             ; value to POKE in .A.
0C30  8E 00 D6   WR80CO   STX   VDCADR       ; ask for an audience
0C33  AE 00 D6   LOOP2    LDX   VDCADR       ; check whether we can get in
0C36  10 FB               BPL   LOOP2        ; not yet, branch back
0C38  8D 01 D6            STA   VDCDAT       ; store the character
0C3B  60                  RTS
0C3C  01 02 03   MESSAGE  .BYTE 1,2,3,4,5
0C41  00                  .BYTE 0
```

See also CUST80, VDCCOL.

Name

Read bytes from a sequential or program file into a buffer

Description

READBF, with the aid of three routines—**OPENFL**, **READFL**, and **CLOSFL**—reads in either a sequential file or a program file from disk and stores it in a data buffer. The address of this buffer is passed from the calling program in the X (low byte) and Y (high byte) registers.

Prototype

In the calling program (MAIN below):

1. Define the address of the data buffer (as BUFFER) in the equates.
2. On the 128, set the bank to 15. On both machines, load the buffer address in .X (low byte) and .Y (high byte). Then JSR to **READBF**.

In **READBF** itself:

1. Store the buffer address in .X and .Y to zero page.
2. Open a sequential or program filename with **OPENFL**.
3. Read in data from the open file into the buffer using **READFL**.
4. Close the open file with **CLOSFL**. Return the ending address of the file in .X (low byte) and .Y (high byte).

Explanation

The example program reads a sequential file (called SEQUENTIAL) from disk into a buffer located at 16384. To read in a program file, change the suffix on the filename from *,S,R* to *,P,R*.

To locate the incoming file data at a location other than 16384, simply change the buffer address (BUFFER) in the equates. Alternatively, you could change the LDX and LDY at the very start of the framing routine.

READBF itself is a short routine (the various support routines for opening, reading, and closing the file take up most of the space). The X and Y registers containing the buffer address are first stored to a free location in zero page (ZP). The three routines **OPENFL**, **READFL**, and **CLOSFL** are then called to read in the file. Before returning to the main program, the ending address of the file is stored in the X (low byte) and Y (high byte) registers.

This routine is a good example of modular programming. The main routine calls **READBF**, which in turn calls three in-

dependent subroutines for opening, reading, and closing a file. If you want to read a file and print it to the screen, add another JSR to the main routine. If you want to alphabetize, just append the appropriate subroutine to the end of the program and stick a JSR in the main routine. By writing the program in small, easy-to-handle modules, you will retain a lot of flexibility.

Note: You can add disk error checking to this program by including **DERRCK** at the places marked in the source code.

Routine

```
C000                SETLFS   =    65466
C000                SETNAM   =    65469
C000                OPEN     =    65472
C000                CHKIN    =    65478
C000                CHRIN    =    65487
C000                CLOSE    =    65475
C000                CLRCHN   =    65484
C000                STATUS   =    144
C000                ZP       =    251
C000                BUFFER   =    16384      ; starting address where incoming data will
                                             ; be stored
C000                                         ; SETBNK = 65384; Kernal bank number for
                                             ; data and filename (128 only)
C000                                         ; MMUREG = 65280; MMU configuration
                                             ; register (128 only)
                                             ;
                                             ; READBF uses the following three routines
                                             ; to read characters
                                             ; OPENFL to open the sequential/program
                                             ; file
                                             ; READFL to read in characters from the file
                                             ; CLOSFL to close the file and restore the
                                             ; default input device
                                             ;
C000                MAIN     =    *
                                             ; LDA #0, set bank 15 (128 only)
                                             ; STA MMUREG; (128 only)
C000  A2 00                  LDX  #<BUFFER   ; low byte of buffer address
C002  A0 40                  LDY  #>BUFFER   ; and high byte
C004  20 08 C0               JSR  READBF     ; go read data from file
C007  60                     RTS
                                             ;
                                             ; READBF opens a SEQ or PRG file and
                                             ; reads all data into a buffer.
                                             ; Enter with address of storage buffer in .X
                                             ; (low) and .Y (high).
                                             ; Upon return, .X and .Y will hold the end-of-
                                             ; buffer address.
C008  86 FB         READBF   STX  ZP         ; store low byte of storage buffer
C00A  84 FC                  STY  ZP+1       ; store high byte also
C00C  20 1C C0               JSR  OPENFL     ; open file
C00F  20 32 C0               JSR  READFL     ; read data from open file and store in
                                             ; buffer
C012  A9 01                  LDA  #1         ; file 1
C014  20 49 C0               JSR  CLOSFL     ; close file and restore default devices
C017  A6 FB                  LDX  ZP         ; low byte of end-of-file address
C019  A4 FC                  LDY  ZP+1       ; high byte of address for EOF
C01B  60                     RTS             ; return to MAIN
                                             ;
```

```
                                              ; OPENFL opens a sequential or program file
                                              ; with for reading/writing.
C01C              OPENFL   =     *
                                              ;
                                              ; Open channel 15 here if you include error
                                              ; checking (DERRCK)
                                              ;
C01C  A9 01                LDA   #1           ; logical file 1
C01E  A2 08                LDX   #8           ; device number for disk drive
C020  A0 02                LDY   #2           ; secondary address (2-4 is okay)
C022  20 BA FF             JSR   SETLFS       ; set file to be opened
                                              ;
                                              ; Include the following three instructions on
                                              ; the 128 only.
                                              ; LDA BNKNUM; bank number for data
                                              ; LDX BNKFNM, bank containing filename
                                              ; JSR SETBNK
C025  A9 10                LDA   #FNLENG      ; length of filename
C027  A2 4F                LDX   #<FILENM     ; address of filename
C029  A0 C0                LDY   #>FILENM
C02B  20 BD FF             JSR   SETNAM       ; set up filename
C02E  20 C0 FF             JSR   OPEN         ; open the file for reading
                                              ;
                                              ; JSR DERRCK; insert for disk error checking
                                              ;
C031  60                   RTS                ; return to READBF
                                              ;
                                              ; READFL reads characters from a sequential
                                              ; or program file
                                              ; and stores them in a buffer whose address
                                              ; is in zero page.
C032  A2 01       READFL   LDX   #1
C034  20 C6 FF             JSR   CHKIN        ; take input from file 1
C037  A0 00                LDY   #0           ; index into the storage buffer
C039  20 CF FF    RDLOOP   JSR   CHRIN        ; get a byte from open file
C03C  91 FB                STA   (ZP),Y       ; put it in the storage buffer
C03E  E6 FB                INC   ZP           ; increment low byte of buffer address
C040  D0 02                BNE   STATCK       ; low byte hasn't rolled over, so skip forward
C042  E6 FC                INC   ZP+1         ; otherwise, increase high byte
                                              ;
                                              ; STATCK checks the I/O status flag for end
                                              ; of file
C044  A5 90       STATCK   LDA   STATUS       ; check for EOF
C046  F0 F1                BEQ   RDLOOP       ; a zero indicates there is more remaining, so
                                              ; continue reading
C048  60                   RTS                ; return to READBF
                                              ;
                                              ; CLOSFL closes the logical file specified in
                                              ; .A and restores default devices.
C049  20 C3 FF    CLOSFL   JSR   CLOSE        ; close file in .A
C04C  4C CC FF             JMP   CLRCHN       ; clear all channels, restore default devices,
                                              ; and RTS
                                              ;
                                              ; insert DERRCK routine here if you're
                                              ; including error checking.
                                              ;
C04F  30 3A 53    FILENM   .ASC  "0:SEQUENTIAL,S,R"
                                              ; example sequential file to read
                                              ; ,S,R is optional when reading sequential
                                              ; files
                                              ; Change to "0:PROGRAM,P,R" to read a
                                              ; program file.
```

```
C05F    FNLENG  =   * FILENM   ; length of filename
                               ;
                               ; Include the next two variables on the 128
                               ; only
                               ; BNKNUM .BYTE 0; bank number where
                               ; data is to be stored
                               ; BNKFNM .BYTE 0; bank number where
                               ; ASCII filename is located
```

See also OPENFL, READFL.

Name

Read characters from a sequential or program file

Description

With **READFL**, you can read characters into memory from either a sequential or a program disk file. The routine stores this incoming data in a buffer named by a zero-page pointer.

Prototype

1. Before accessing **READFL**, call **OPENFL** to open a channel from which to read data.
2. Define the input channel as the one opened with Kernal CHKIN.
3. Read bytes one at a time from this channel, storing them in a memory buffer using zero-page addressing.
4. Check the status flag (STATUS) for the last byte in the incoming file.
5. If STATUS is zero, continue reading bytes. Otherwise, RTS to the calling program.

Explanation

The subroutine below is not a complete program; it's designed to be used in conjunction with several other subroutines. (See the complete program under **READBF**, which reads a file into a buffer.) Before coming into **READFL**, you must do two things—open an input channel with **OPENFL** and store the address of the memory buffer into zero page.

Once in **READFL**, data is continuously read until the STATUS flag at location 144 contains a nonzero value. When this occurs, the routine returns to the calling program.

Note: The routine as written takes input from logical file 1. To read in data from another channel, load the appropriate channel number into the X register at $C000–$C001.

Routine

```
C000                  CHKIN    =     65478
C000                  CHRIN    =     65487
C000                  STATUS   =     144
C000                  ZP       =     251
                                                 ;
                                                 ; READFL reads characters from a sequential
                                                 ; or program file and
                                                 ; stores them to a buffer whose address is in
                                                 ; zero page
C000  A2 01           READFL   LDX   #1
C002  20 C6 FF                 JSR   CHKIN       ; take input from file 1
C005  A0 00                    LDY   #0          ; index into the storage buffer
C007  20 CF FF        RDLOOP   JSR   CHRIN       ; get a byte from open file
```

```
C00A  91  FB               STA   (ZP),Y      ; put it in the storage buffer using zero-
                                             ; page addressing
C00C  E6  FB               INC   ZP          ; increment low byte of buffer address
C00E  D0  02               BNE   STATCK      ; low byte hasn't rolled over, so skip
                                             ; forward
C010  E6  FC               INC   ZP+1        ; otherwise, increase high byte
                                             ;
                                             ; STATCK checks the I/O status flag for
                                             ; end-of-file.
C012  A5  90      STATCK   LDA   STATUS      ; check for EOF
C014  F0  F1               BEQ   RDLOOP      ; a zero indicates there is more remaining,
                                             ; so continue reading
C016  60                   RTS               ; return to main program
```

See also OPENFL, READBF.

Name

Rename a disk file

Description

This routine renames a file by opening channel 15 and sending the command "R0:*newname*=0:*oldname*". You may note that it's very similar in structure to the other DOS commands.

Prototype

1. Open the disk command channel (SETLFS, SETNAM, OPEN).
2. Provide the rename command as the filename in SETNAM.
3. Close things up.

Explanation

The rename command is provided in the data area at the end of the routine. If you were to use this example program yourself, you'd probably want build the command from an old name and new name requested from the user.

Routine

```
C000                SETLFS   =     $FFBA
C000                SETNAM   =     $FFBD
C000                OPEN     =     $FFC0
C000                CLOSE    =     $FFC3
C000                CLRCHN   =     $FFCC
                                                ;
C000  A9 01         RENAME   LDA   #1           ; logical file number
C002  A2 08                  LDX   #8           ; device number for disk drive
C004  A0 0F                  LDY   #15          ; secondary address for drive command
                                                ; channel
C006  20 BA FF               JSR   SETLFS       ; prepare to open it
C009  A9 15                  LDA   #BUFLEN      ; length of buffer
C00B  A2 1E                  LDX   #<BUFFER     ; .X and .Y hold the
C00D  A0 C0                  LDY   #>BUFFER     ; address of the buffer
C00F  20 BD FF               JSR   SETNAM       ; set up command as name
C012  20 C0 FF               JSR   OPEN         ; open it
C015  A9 01                  LDA   #1           ; and immediately
C017  20 C3 FF               JSR   CLOSE        ; close the command channel
C01A  20 CC FF               JSR   CLRCHN       ; clear the channels
C01D  60                     RTS                ; all done
                                                ; Data area
                             .ASC  "R0:NEWNAME=0:OLDNAME"
                                                ; substitute your own filenames here
C032  0D                     .BYTE 13           ; RETURN character
C033                BUFLEN   =     * - BUFFER
```

See also CONCAT, COPYFL, FORMAT, INITLZ, SCRTCH, VALIDT.

Name

Simple renumber routine (line numbers only)

Description

Changing the line numbers of a BASIC program is relatively easy. What's difficult is revising the GOTOs, GOSUBs, and other references within the various lines. This routine changes only the actual line numbers; the other references remain as they were.

Prototype

1. Using two zero-page locations, set up a pointer to the beginning of the BASIC line.
2. Load the line link, which points to the next line in memory. If the line link contains two zeros, exit the routine.
3. Copy the desired line number into the current line.
4. Update the line number, adding the STEP value.
5. Copy the line link to the first zero-page location and loop back to step 2.

Explanation

Before the text of a BASIC line in memory, there are four bytes—two 2-byte pointers. The first is the line link that points to the beginning of the next line (which, in turn, points the next line link, and so on, to the end of the program). The next two bytes provide the line number in low-byte/high-byte format.

A pointer at location 43 (location 45 on the 128) contains the address of the beginning of the BASIC program. The end of the BASIC program is marked by a line link of $0000.

To renumber, get the TXTTAB pointer and copy it to a zero-page location (Z2, in the example). The main loop starts by copying the contents of Z2 to Z1. Then, .Y is loaded with a 0 and a 1, and the next line link is copied indirectly from Z1 to Z2. Finally, .Y is increased to 2 and then to 3 (to point to the line number in memory), and the desired line number is stored in memory.

The line number is incremented by the STEP value, and the process repeats. As soon as a line link of $0000 is discovered, the program ends and the renumbering is complete.

Note: To ensure that this routine works properly on the 128, enter the BASIC line **BANK 0** before you SYS to the program. Unlike most other programs, which have to be in bank

15 to be able to call Kernal routines, this routine needs to be in bank 0.

Routine

```
C000                TXTTAB    =     43           ; TXTTAB = 45 on the 128
C000                Z1        =     $FB
C000                Z2        =     $FD
                                                 ;
C000  4C  09  C0              JMP   RENUM1       ; jump around the table
                                                 ;
C003  14  00        FIRST     .BYTE 20,0         ; first line number
C005  0A  00        STEP      .BYTE 10,0         ; renumber by tens
C007  00  00        CURRENT   .BYTE 0,0          ; current line number
                                                 ;
C009  A2  01        RENUM1    LDX   #1           ; do some copying
C00B  B5  2B        COPY      LDA   TXTTAB,X     ; the start of BASIC text
C00D  95  FD                  STA   Z2,X         ; goes into Z2
C00F  BD  03  C0              LDA   FIRST,X      ; and the line number
C012  9D  07  C0              STA   CURRENT,X    ; goes into CURRENT
C015  CA                      DEX
C016  10  F3                  BPL   COPY         ; loop back
                                                 ;
C018  20  2B  C0    BEGIN     JSR   CPZ2Z1       ; copy the pointer from Z2 to Z1
C01B  20  34  C0              JSR   LLINK        ; and set up the line link for the next line
                                                 ; in Z2
C01E  A5  FB                  LDA   Z1           ; two zeros
C020  05  FC                  ORA   Z1+1         ; in Z1
C022  D0  01                  BNE   AHEAD        ; mean that
C024  60                      RTS                ; we're done and can quit
                                                 ;
C025  20  41  C0    AHEAD     JSR   RENLIN       ; else renumber the line
C028  4C  18  C0              JMP   BEGIN        ; and go back for another
                                                 ;
C02B  A5  FD        CPZ2Z1    LDA   Z2           ; copy Z2
C02D  85  FB                  STA   Z1           ; to Z1
C02F  A5  FE                  LDA   Z2+1         ; high byte, too
C031  85  FC                  STA   Z1+1         ; and
C033  60                      RTS                ; that's all
                                                 ;
C034  A0  00        LLINK     LDY   #0           ; get Z2 ready
C036  B1  FB                  LDA   (Z1),Y       ; low byte
C038  85  FD                  STA   Z2           ; into Z2
C03A  C8                      INY
C03B  B1  FB                  LDA   (Z1),Y       ; high byte
C03D  85  FE                  STA   Z2+1         ; into Z2+1; now Z2 is ready for the next
                                                 ; line
C03F  C8                      INY                ; INY one more time, so it's 2
C040  60                      RTS                ; go back
                                                 ;
C041                RENLIN    =     *            ; remember, .Y is now 2, from LLINK above
C041  AD  07  C0              LDA   CURRENT      ; low byte of CURRENT
C044  91  FB                  STA   (Z1),Y       ; into the program
C046  AD  08  C0              LDA   CURRENT+1    ; high byte
C049  C8                      INY
C04A  91  FB                  STA   (Z1),Y       ; also
                                                 ;
```

```
C04C  18           CLC                 ; now add the STEP to CURRENT
C04D  AD 05  C0    LDA   STEP
C050  6D 07  C0    ADC   CURRENT       ; add it
C053  8D 07  C0    STA   CURRENT       ; store it
C056  AD 06  C0    LDA   STEP+1        ; high byte
C059  6D 08  C0    ADC   CURRENT+1     ; add
C05C  8D 08  C0    STA   CURRENT+1     ; save
C05F  60           RTS                 ; and that's that
```

See also DATAMK.

Name

Generate a random floating-point number using BASIC's RND(1) function

Description

Random integer values can be generated with **RNDBYT** (one-byte) or **RDBYRG** (two-byte). At times, though, you may wish to generate a random floating-point number.

RND1VL uses BASIC's own RND function to produce a random floating-point number in the range 0–0.999999999. You can place this number in any numeric range, just as if you were in BASIC, by multiplying it and adding some base value. For instance, if you needed floating-point numbers from 5.0 through 15.0, you would multiply the number returned by **RND1VL** by 10 and add 5.

Prototype

JMP into BASIC's RND function to cause a random value from 0 through 0.999... to be placed in floating-point accumulator 1 (FAC1).

Explanation

Ten random floating-point numbers in the range 0–0.999... are generated by the example program and printed to the screen.

A random number is first placed in floating-point accumulator 1 by **RND1VL**. Using FOUT, the contents of FAC1 are converted to an ASCII string and are stored in the workspace area at the top of the stack (beginning at $100). Finally, with **FACPRT**, the string within the workspace is printed to the screen. This process is repeated for each of the ten values.

RND1VL itself is very short. In it, we jump midway into BASIC's RND function routine at 57534 on the 64 (33877 on the 128). This causes a random floating-point number to be transferred from the seed value in RNDX (location 139 on the 64 or location 4635 on the 128) to FAC1.

Routine

```
C000        CHROUT  =   65490
C000        FAC1    =   97          ; FAC1 = 99 on the 128
C000        FOUT    =   48605       ; FOUT = 36418 on the 128—converts FAC1
                                    ; to ASCII
C000        STWORK  =   256         ; workspace at top of the stack
```

```
C000                RND1     =     57534      ; RND1 = 33877 on the 128; RND(1)
                                              ; function
                                              ;
                                              ; Generate ten numbers (0-0.999...) using the
                                              ; RND(1) function and print them.
C000  A2 0A                  LDX   #10        ; initialize counter X to give ten random
                                              ; numbers
C002  8E 2D C0               STX   TEMPX      ; save X
C005  20 1C C0  LOOP         JSR   RND1VL     ; get random number using RND(1)
C008  20 DD BD               JSR   FOUT       ; convert contents of FAC1 to ASCII string
                                              ; string is in stack area
C00B  20 1F C0               JSR   FACPRT     ; print the FAC1
C00E  A9 0D                  LDA   #13        ; print RETURN
C010  20 D2 FF               JSR   CHROUT
C013  CE 2D C0               DEC   TEMPX      ; decrement counter
C016  AE 2D C0               LDX   TEMPX      ; and put in .X for branch
C019  D0 EA                  BNE   LOOP       ; if we haven't done all ten, continue
C01B  60                     RTS
                                              ;
                                              ; RND1VL fetches a random number using
                                              ; RND(1) and places it in FAC1
C01C  4C BE E0  RND1VL       JMP   RND1       ; get random number
                                              ;
                                              ; FACPRT prints the number in floating-
                                              ; point accumulator 1.
C01F  A0 00     FACPRT       LDY   #0         ; as an index
C021  B9 00 01  MORE         LDA   STWORK,Y   ; load each ASCII byte of string
C024  F0 06                  BEQ   OUT        ; if zero byte, we're finished
C026  20 D2 FF               JSR   CHROUT     ; print it
C029  C8                     INY              ; next byte
C02A  D0 F5                  BNE   MORE       ; branch always
C02C  60        OUT          RTS
                                              ;
C02D  00        TEMPX        .BYTE 0          ; temporary storage for .X
```

See also RD2BYT, RDBYRG, RNDBYT.

Name

Generate a random one-byte integer value (0–255)

Description

Many programs, especially games and educational programs, require randomness. Often, what is called for is a one-byte random integer in the range 0–255. This routine lets you generate such a number from the random oscillations of the noise waveform.

Prototype

In an initialization routine (RDINIT):

1. Set voice 3 to a high frequency.
2. Select the noise waveform.
3. Turn off the SID chip volume and disconnect the output of voice 3.

In **RNDBYT** itself:

4. Take a random byte value from voice 3's random number generator (RANDOM) and return it in .A.

Explanation

In the example program, an interesting visual effect is created by repeatedly placing a random color value somewhere in the first 256 bytes of screen color RAM. Pressing any key exits the routine.

RNDBYT is actually a two-part routine. In the first part, labeled RDINIT, voice 3 of the SID chip is initialized so as to generate random numbers in RANDOM (location 54299). This is done by setting the high byte of the frequency register for voice 3 (FREHI3) to 255 and selecting the noise waveform by setting bit 7 of voice 3's control register (VCREG3). Since we don't want to actually hear the noise, we turn off the SID chip volume and disconnect the audio output of voice 3 by storing a 128 to SIGVOL, the volume and filter select register. Selecting a frequency value high byte of 255 insures that the values in RANDOM change very rapidly.

RDINIT need be accessed only once early in your main program. After that, you can take random values as needed from RANDOM. This is exactly what **RNDBYT** does, returning the random byte in the accumulator.

Routine

```
C000                GETIN   =     65508
C000                COLRAM  =     55296        ; start of screen color memory
C000                FREHI3  =     54287        ; voice 3 frequency control register (high
                                               ; byte)
C000                VCREG3  =     54290        ; voice 3 control register
C000                SIGVOL  =     54296        ; volume and filter select register
C000                RANDOM  =     54299        ; oscillator 3/random number generator
                                               ;
                                               ; Generate a random byte value from SID
                                               ; chip voice 3.
                                               ; Put a random color anywhere in first 256
                                               ; bytes of screen.
                                               ; Quit when any key is pressed.
C000  20 13 C0  MAIN        JSR   RDINIT       ; initialize SID voice 3 for random numbers
C003  20 21 C0  LOOP        JSR   RNDBYT       ; get a random byte for screen offset
C006  A8                    TAY                ; store offset in .Y
C007  20 21 C0              JSR   RNDBYT       ; get random number for color byte
C00A  99 00 D8              STA   COLRAM,Y     ; store color byte randomly in first quarter
C00D  20 E4 FF              JSR   GETIN        ; check for a keypress
C010  F0 F1                 BEQ   LOOP         ; no keypress, so continue
C012  60                    RTS                ; else, quit
                                               ;
                                               ; Routine to initialize SID voice 3 for random
                                               ; numbers
C013  A9 FF     RDINIT      LDA   #$FF         ; set voice 3 frequency (high byte) to
                                               ; maximum
C015  8D 0F D4              STA   FREHI3
C018  A9 80                 LDA   #%10000000
C01A  8D 12 D4              STA   VCREG3       ; select noise waveform and start release for
                                               ; voice 3
C01D  8D 18 D4              STA   SIGVOL       ; turn off volume and disconnect output of
                                               ; voice 3
C020  60                    RTS
                                               ;
                                               ; RNDBYT returns a random byte value
                                               ; in .A.
C021  AD 1B D4  RNDBYT      LDA   RANDOM       ; get single-byte random number
C024  60                    RTS
```

See also RD2BYT, RDBYRG, RND1VL.

Name

Set the repeat key flag

Description

In certain applications, such as a word processor or a game featuring keyboard control, you'll need to let the keys repeat. But at other times you'll want to fetch only one keypress at a time.

For instance, suppose you need to ask the user a series of questions. If keypresses can repeat, and if the user lets a finger tarry on the RETURN key, several questions can easily be skipped before the user realizes what is happening. By storing a 64 in the repeat flag (RPTFLG), you can prevent this situation.

Prototype

1. Define RPSTAT as 0, 64, or 128.
2. Load and store RPSTAT in the repeat flag.

Explanation

The accompanying program makes all keypresses nonrepeating.

Note: The repeat flag (RPTFLG) is located at 650 on the 64 and at 2594 on the 128. It can contain either a 0, a 64, or a 128. A value of 0 causes only certain keys to repeat (specifically the cursor keys, the INST/DEL key, and the space bar). As illustrated, a value of 64 prevents all keys from repeating, while 128 allows all keys to repeat.

The default value for this location is different on the 64 and the 128. On the 64, it's 0; on the 128, the default value is 128.

Routine

```
C000              RPTFLG   =     650      ; RPTFLG = 2594 on the 128—repeat key
                                          ; flag
                                          ;
                                          ; Disable all repeats.
C000  AD 07  C0   RPTKEY   LDA   RPSTAT
C003  8D 8A  02            STA   RPTFLG
C006  60                   RTS
                                          ;
C007  40          RPSTAT   .BYTE 64       ; disable all repeats
                                          ; 0 allows certain cursor keys to repeat.
                                          ; 128 enables all repeats
```

Name

Restore registers from memory

Description

After using **SVREGM** to save the registers to memory, you can get them back with **RSREGM.**

Prototype

1. Load the processor status (.P) and push it onto the stack.
2. Load the .A, .X, and .Y registers from memory.
3. Pull .P (PLP) from the stack.

Explanation

Operations such as loading from memory (LDA, LDX, and LDY) affect both the zero and the minus flags in the processor status .P, so .P must be the last register restored. Since there's no direct way to load .P from memory, the previously saved register must be pushed onto the stack by .A and then pulled with the PLP instruction. Apart from this one little shuffling step, the rest of the routine is short and straightforward.

Routine

```
C000  AD 12  C0  RSREGM   LDA   TEMPP     ; first get the .P status register
C003  48                  PHA             ; push it temporarily
C004  AD 0F  C0           LDA   TEMPA     ; get .A
C007  AE 10  C0           LDX   TEMPX     ; get .X
C00A  AC 11  C0           LDY   TEMPY     ; get .Y
C00D  28                  PLP             ; get .P from the stack (where it was
                                          ; pushed from .A)
C00E  60                  RTS             ; we're done
                                          ;
                                          ; variables
C00F  00         TEMPA    .BYTE 00        ; note—these were
C010  00         TEMPX    .BYTE 00        ; put in place by the
C011  00         TEMPY    .BYTE 00        ; SVREGM routine
C012  00         TEMPP    .BYTE 00
```

See also SVREGM, SVREGS.

Name

Restore all Kernal indirect vectors

Description

RSTVEC reinitializes the 16 Kernal vectors in RAM beginning at location 788 to their default warm start values. This routine is useful in situations where you have altered these vectors—so that they point to your own RAM-based routines—and later want to change them back en masse.

Prototype

1. Disable IRQ interrupts with an SEI.
2. JSR to the Kernal RESTOR routine, reenable IRQ interrupts with a CLI, and RTS to your calling program.

Explanation

RSTVEC relies on the Kernal routine RESTOR to reset the interrupt and Kernal I/O (Input/Output) vectors at locations 788–819. Since the IRQ interrupt vector is among those being restored, it's best to prevent any IRQ interrupts from being serviced while you're changing these vectors. This is accomplished here with an SEI prior to calling RESTOR.

For an example of how to use **RSTVEC** in your own programs, take a look at **ALARM2**. This routine sets the alarm for the second time-of-day clock. When the alarm goes off, an NMI interrupt occurs. At this point, we completely disable the alarm function with **RSTVEC**.

You might note that the RESTOR routine is normally accessed when either a cold or a warm start is carried out (see **COLDST** and **WARMST**). In both instances, the Kernal indirect vectors are reset.

The same cannot be said of the BASIC indirect vectors. This series of vectors, occupying locations 768–779 on the 64 (768–785 on the 128), are reinitialized only during the cold-start procedure. You can reset the BASIC vectors yourself by JSRing to location 58451 in Kernal ROM on the 64 or to 16977 in BASIC ROM on the 128.

Routine

```
C000              RESTOR   =     65418     ; Kernal routine to restore I/O RAM vectors
                                           ; to default values
                                           ;
C000  78          RSTVEC   SEI             ; disable IRQ interrupts while resetting
                                           ; IRQ vector
C001  20  8A  FF           JSR   RESTOR    ; reset page 3 RAM vectors to ROM table
                                           ; values
C004  58                   CLI             ; reenable IRQ interrupts
C005  60                   RTS             ; we're done
```

See also DISRSR, DISTOP, ERRRDT.

Name

Save a BASIC program

Description

SAVEBS saves a BASIC program to disk, regardless of where the BASIC workspace is located at the time of the save.

Prototype

1. On the 128, set the bank to 15.
2. Set up the parameters as 1,8,0 for a save (SETLFS, SETNAM).
3. On the 128, call SETBNK to specify the bank containing the program you intend to save and the bank containing its filename.
4. Load .A with the address of TXTTAB (the location of the zero-page pointer to the start of BASIC text).
5. Load .X and .Y with the values in end-of-BASIC text pointer.
6. JSR to SAVE.

Explanation

SAVEBS, relying on several Kernal routines, saves a copy of the contents of the BASIC program text area to disk. As with all saves, a secondary address of zero is required.

Before executing SAVE, we set the zero-page pointer to the start of BASIC text (TXTTAB) in the accumulator. The X and Y registers are loaded with the two-byte ending address of the BASIC program at VARTAB. On the 128, replace VARTAB with TEXTTP.

To use this routine to save your own BASIC programs, substitute for "BASIC PROGRAM" the name of the program you wish to save.

Note: **SAVEBS** currently lacks disk error checking. You can add this feature if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM as noted in the source listing. Jump to **DERRCK** immediately after the JSR SAVE instruction. Furthermore, be sure to open the error channel (15) at the beginning of the program (also noted in the source listing).

On the 128, include BNKNUM and BNKFNM at the end of your program.

Routine

```
C000            SETLFS   =   65466
C000            SETNAM   =   65469
C000            SAVE     =   65496
```

```
C000                TXTTAB   =    43            ; TXTTAB = 45 on the 128—start of BASIC
                                                ; pointer
C000                VARTAB   =    45            ; end-of-BASIC pointer—substitute
                                                ; TEXTTP = 4624 for the 128
C000                                            ; SETBNK = 65384, Kernal bank number for
                                                ; data and filename (128 only)
C000                                            ; MMUREG = 65280, MMU configuration
                                                ; register (128 only)
                                                ;
                                                ; Save a BASIC program to disk.
                                                ;
                                                ; Open channel 15 here if you include disk
                                                ; error checking (DERRCK).
                                                ;
C000                SAVEBS   =    *
                                                ; LDA #0; set bank 15 (128 only)
                                                ; STA MMUREG; (128 only)
C000  A9 01                  LDA  #1            ; logical file 1
C002  A2 08                  LDX  #8            ; device number for disk drive
C004  A0 00                  LDY  #0            ; for all saves
C006  20 BA FF               JSR  SETLFS        ; set for a save
                                                ; Include the following three instructions
                                                ; on the 128 only.
                                                ; LDA BNKNUM; bank number in which
                                                ; program text is located
                                                ; LDX BNKFNM; bank containing the
                                                ; filename
                                                ; JSR SETBNK
C009  A9 0F                  LDA  #FNLENG       ; length of filename
C00B  A2 1C                  LDX  #<FILENM      ; address of filename
C00D  A0 C0                  LDY  #>FILENM
C00F  20 BD FF               JSR  SETNAM        ; set up filename
C012  A9 2B                  LDA  #TXTTAB       ; address of zero-page pointer to the start of
                                                ; the program
                                                ; Change VARTAB in the next two
                                                ; instructions to TEXTTP on the 128.
C014  A6 2D                  LDX  VARTAB        ; low byte for end of BASIC program
                                                ; address
C016  A4 2E                  LDY  VARTAB+1      ; high byte for end of BASIC program
                                                ; address
C018  20 D8 FF               JSR  SAVE          ; save the BASIC file to disk
                                                ;
                                                ; JSR DERRCK; insert for disk error
                                                ; checking
                                                ;
C01B  60                     RTS
                                                ;
                                                ; Insert DERRCK here if you're including
                                                ; error checking.
                                                ;
C01C  30 3A 42      FILENM   .ASC "0:BASIC PROGRAM"
                                                ; substitute your filename here (<=16
                                                ; characters)
C02B                FNLENG   =    *-FILENM      ; length of filename
                                                ; Include the next two variables on the
                                                ; 128 only.
                                                ; BNKNUM .BYTE 0; bank number where
                                                ; program to be saved is located
                                                ; BNKFNM .BYTE 0; bank number where
                                                ; program's filename is located
```

See also SAVEML, VERIFY.

Name

Save an ML program

Description

SAVEML is quite versatile. With it, you can save to disk an ML program or any block of binary data such as sprite patterns, custom characters, hi-res screens, and so on, from any memory location specified.

Prototype

1. On the 128, set the bank to 15.
2. Store the starting address of the ML program (STPROG) in zero page.
3. Set up the parameters for a save (SETLFS, SETNAM).
4. On the 128, prior to SETNAM, load .A with the number of the bank containing the program to be saved and .X with number of the bank containing its filename. Then JSR to SETBNK.
5. Load immediately the zero-page pointer to STPROG.
6. Load .X and .Y with the ending address of the ML program (ENDPRG).
7. JSR to SAVE.

Explanation

The example routine is set up to save an ML program named "ML PROGRAM", which runs from location 49152 (STPROG) through location 50000 (ENDPRG 1), or alternatively, on a 128, to save an ML program residing in memory from 3072 through 3920 (when STPROG and ENDPRG are set in the source listing accordingly). Notice that whether you're on the 64 or 128, you must always add one to the value of the last byte in your code. The SAVE routine saves up to (but not including) the last byte specified.

To save your own ML program, just substitute its filename for "ML PROGRAM" and specify its starting and ending address (plus 1) as STPROG and ENDPRG, respectively, in the equates. Furthermore, the secondary address, when the file parameters are set up, must contain a zero for all saves.

Note: **SAVEML** currently lacks disk error checking. You can add this feature if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM, as noted in the source listing. Jump to **DERRCK** immediately after the JSR SAVE instruction. Be sure to open the error chan-

nel (15) at the beginning of the program (also noted in the source listing).

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000                SETLFS    =    65466
C000                SETNAM    =    65469
C000                SAVE      =    65496
C000                ZP        =    251
C000                STPROG    =    49152      ; starting address of ML program (perhaps
                                              ; 3072 on 128)
C000                ENDPRG    =    50001      ; ending address of ML program plus 1
                                              ; perhaps 3921 on 128)
C000                SETBNK    =    65384      ; Kernal bank number for SAVE and filename
                                              ; (128 only)
C000                MMUREG    =    65280      ; MMU configuration register (128 only)
                                              ;
                                              , Save an ML program from 49152 through
                                              ; 50000 (3072-3920 on the 128)
                                              , Open channel 15 here if you include disk
                                              ; error checking (DERRCK).
                                              ;
C000                SAVEML    =    *
                                              ; LDA #0; set the 128 to bank 15 (128 only)
                                              ; STA MMUREG; (128 only)
C000  A2 00                   LDX  #<STPROG   ; low byte of program address
C002  86 FB                   STX  ZP         ; store in zero-page
C004  A0 C0                   LDY  #>STPROG   ; high byte of program address
C006  84 FC                   STY  ZP+1       ; also store in zero-page
C008  A9 01                   LDA  #1         ; logical file number (value doesn't matter)
C00A  A2 08                   LDX  #8         ; device number for disk drive
C00C  A0 00                   LDY  #0         , secondary address for all saves
C00E  20 BA FF                JSR  SETLFS     ; set parameters for save
                                              ; Include the following three instructions
                                              , for the 128 only.
                                              ; LDA BNKNUM; bank containing the
                                              ; program
                                              ; LDX BNKFNM; bank containing the
                                              ; ASCII filename
                                              ; JSR SETBNK
C011  A9 0C                   LDA  #FNLENG    ; length of filename
C013  A2 24                   LDX  #<FILENM   ; address of filename
C015  A0 C0                   LDY  #>FILENM
C017  20 BD FF                JSR  SETNAM     ; set up filename
C01A  A9 FB                   LDA  #ZP        ; zero-page pointer to start of ML program
C01C  A2 51                   LDX  #<ENDPRG   ; low-byte address for end of ML program
C01E  A0 C3                   LDY  #>ENDPRG   ; high-byte address for end of ML program
C020  20 D8 FF                JSR  SAVE       ; save the ML file
                                              ;
                                              ; JSR DERRCK; Insert here for disk error
                                              ; checking
                                              ;
```

```
C023  60                  RTS
                                                 ;
                                                 ; Insert DERRCK here if you're including
                                                 ; error checking.
                                                 ;
C024  30  3A  4D  FILENM  .ASC  "0:ML PROGRAM"
                                                 ; Substitute your own filename here (<=16
                                                 ; characters)
C030              FNLENG  =     * - FILENM       ; length of filename
                                                 ; Include the next two variables on the 128.
                                                 ; BNKNUM .BYTE 0; bank number where
                                                 ; data to be saved is located
                                                 ; BNKFNM .BYTE 0; bank number where
                                                 ; ASCII filename is located
```

See also SAVEBS, VERIFY.

Name

Convert screen codes to Commodore ASCII characters

Description

Commodore computers, including the 64 and the 128, represent characters in different ways. When characters are printed (with CHROUT), they are represented by Commodore ASCII codes. When they are stored directly to screen memory (with STA), so-called screen codes are used. Fortunately, there are some patterns between the two sets of codes. As a result, the actual conversion routine can be relatively short.

You'll probably find a number of uses for **SCRCAS**. Many word processing programs (COMPUTE!'s *SpeedScript* and Pro-Line's *WordPro*, among others) store characters in their files in the form of screen codes. At some point, you may wish to examine the contents of a file that's in screen-coded format by printing it to the screen. Or you may simply want to print portions of screen memory elsewhere on the screen. In either case, a routine like **SCRCAS** is ideal.

Prototype

1. CMP the screen code in .A with zero, setting the N flag if the code is greater than 127.
2. Store the processor status register on the stack.
3. AND with 127, giving a screen code from 0 through 127.
4. Determine in which range of values the screen code lies (0–31, 32–63, 64–95, or 96–127) and flip the necessary bit(s).
5. Restore the N flag with PLP and RTS.

Explanation

The example program converts characters within a file that's been saved in screen-coded format to Commodore ASCII and prints them to the screen.

This is really a two-step process. First, the file (entitled SCREEN CODES) is loaded into a buffer (LOADAD) on an even-page boundary by using **LOADRL**. Each code within the buffer is then converted to a Commodore ASCII character with **SCRCAS** and is printed.

In order to see the program in action, you'll need to initially create a file containing screen codes. As we've suggested, you can do this with *SpeedScript* or with any other program that saves in this format. Change the ASCII string in FILENM

to match the filename you've chosen. Then run the program, changing LOADAD if you wish.

SCRCAS performs the conversion based on the particular range in which the screen code resides. The second half of the screen code set is identical to the first. The only difference is that characters above 127 are in reverse. If the screen code passed in .A exceeds 127, **SCRCAS** sets the N flag to indicate that the character is in reverse. So, upon returning from the routine, you can print the {RVS ON} character—CHR$(18)—if you wish, before printing the actual converted character.

All codes coming to the routine are ANDed with 127 and are handled as if they were in the lower half of the set. Once this has been done, **SCRCAS** determines in which range the screen code lies, with the aid of the table UPPLIM. There are four ranges—0–31, 32–63, 64–95, and 96–127—each sharing similarities in their bit patterns. These similarities make conversion possible.

This setup is best represented in a table where the bit patterns of characters in each range are shown before and after the conversion:

	Before:		**After:**	
Range	**Bit Pattern**	**Range**	**Bit Pattern**	
0–31	%000*x xxxx*	64–95	%010*x xxxx*	
32–63	%001*x xxxx*	32–63	%001*x xxxx*	(the same)
64–95	%010*x xxxx*	96–127	%011*x xxxx*	
96–127	%011*x xxxx*	160–191	%101*x xxxx*	

Within each bit pattern, a 0 designates bits that are always off, and a 1, bits that are always on. The *x* represents bits that may be on or off.

Converting a screen code in the range 0–31 to the range 64–95 requires that you flip bit 6. The second range stays the same. To go from the range 64–95 to the range 96–127, you turn on bit 5. Screen codes within the final range require that both bits 6 and 7 be flipped.

This is exactly what occurs within **SCRCAS**. A lookup table of values (FLIPTB) is used with EOR to convert a particular screen code. So, the routine returns an equivalent Commodore ASCII value in .A with the N flag set for reverse characters.

Note: Since **SCRCAS** corrupts .Y, you should save it to some temporary location (as is done in the example program) before entering **SCRCAS**.

Also, if you're using a 128 with this program, be sure to replace the instruction at PRTLOP with the three instructions following it. This enables the 128 to access the incoming screen codes stored in bank 0. The Kernal routine INDFET is used for this task. INDFET performs an LDA (zero page),Y from within the bank indicated by the X register.

Routine

```
C000                CHROUT   =    65490
C000                SETLFS   =    65466
C000                SETNAM   =    65469
C000                LOAD     =    65493
C000                LOADAD   =    16384       ; buffer for incoming file, positioned on even
                                              ; page boundary
C000                ZP       =    251
C000                SETBNK   =    65384       ; Kernal bank number for LOAD and
                                              ; filename (128 only)
C000                INDFET   =    65396       ; Kernal routine to fetch a byte from any
                                              ; bank (128 only)
                                              ;
                                              ; LOAD a file containing screen codes,
                                              ; convert them to Commodore ASCII
                                              ; characters, and print them
                                              ;
C000  A9 00                  LDA  #<LOADAD    ; store buffer address in zero page
C002  85 FB                  STA  ZP
C004  A0 40                  LDY  #>LOADAD
C006  84 FC                  STY  ZP+1
C008  20 61 C0               JSR  LOADRL      ; LOAD in the file at 16384
C00B  8E 8B C0               STX  EOF         ; LOADRL returns end-of-file address in .X
                                              ; and .Y
C00E  8C 8C C0               STY  EOF+1       ; store these in temporary locations
C011  A9 0D                  LDA  #13         ; print a RETURN
C013  20 D2 FF               JSR  CHROUT
                                              ;
C016  A0 00                  LDY  #0          ; as an index in PRTLOP
C018  8C 8D C0               STY  MAXIMY      ; save .Y
C01B  F0 02                  BEQ  CHKLOP      ; first check whether buffer is less than 256
                                              ; bytes in length
C01D  E6 FC         OUTLOP   INC  ZP+1        ; increment high byte of buffer address
C01F  A5 FC         CHKLOP   LDA  ZP+1        ; see if we're on the last page of buffer
C021  CD 8C C0               CMP  EOF+1
C024  90 0A                  BCC  PRTLOP      ; if not, print a full page
C026  D0 1E                  BNE  EXIT        ; exit if we're one page beyond the end of the
                                              ; buffer
                                              ; We're on the last page of the buffer,
C028  AD 8B C0               LDA  EOF         ; check the low byte in case buffer ends on
                                              ; an even-page boundary
C02B  F0 19                  BEQ  EXIT        ; if so, exit
C02D  8D 8D C0               STA  MAXIMY      ; otherwise, store last page counter in
                                              ; MAXIMY
C030  B1 FB         PRTLOP   LDA  (ZP),Y      ; get a character from the buffer
                                              ;
                                              ; Replace prior line with next three
                                              ; instructions on the 128.
                                              ; PRTLOP LDX #0; store .X and .A
                                              ; beforehand
                                              ; LDA #ZP
                                              ; JSR INDFET; load (.A),.Y from bank .X
                                              ;
```

```
C032  8C 8E C0           STY   TEMPY      ; since SCRCAS corrupts .Y
C035  20 47 C0           JSR   SCRCAS     ; convert it from screen code to Commodore
                                          ; ASCII (both in .A)
C038  AC 8E C0           LDY   TEMPY      ; restore .Y
C03B  20 D2 FF           JSR   CHROUT     ; print it
C03E  C8                 INY              ; for next character
C03F  CC 8D C0           CPY   MAXIMY     ; have we reached the last byte in the current
                                          ; page (.Y = 0) or
                                          ; the final byte in the last page?
C042  D0 EC              BNE   PRTLOP     ; if not, continue
C044  F0 D7              BEQ   OUTLOP     ; otherwise, check page number
C046  60        EXIT     RTS
                                          ;
                                          ; SCRCAS converts screen codes in .A to
                                          ; Commodore ASCII characters in .A.
                                          ; The N flag is set if character was in reverse
                                          ; video prior to conversion
C047  C9 00     SCRCAS   CMP   #0         ; sets N flag if result is >=128 (if .A
                                          ; >=128)
C049  08                 PHP              ; save N flag status
C04A  29 7F              AND   #127       ; 0-127 and 128-255 are the same, except
                                          ; 128-255 is in reverse video
C04C  A0 04              LDY   #4         ; index goes 3-2-1-0
C04E  88        LOOP     DEY
C04F  D9 59 C0           CMP   UPPLIM,Y   ; is character greater than upper limit
                                          ; value?
C052  B0 FA              BCS   LOOP       ; yes, so check next limit
C054  59 5D C0           EOR   FLIPTB,Y   ; flip corresponding bit(s)
C057  28                 PLP              ; restore N flag (as normal/reverse
                                          ; indicator)
C058  60                 RTS
                                          ;
                                          ; Upper limit plus one of each range and
                                          ; appropriate value to exclusive-OR
C059  80 60 40  UPPLIM   .BYTE 128,96,64,32
C05D  C0 20 00  FLIPTB   .BYTE 192,32,0,64
C061            LOADRL   =     *          ; LOAD a binary file from disk
                                          ;
                                          ; OPEN channel 15 here if you include disk
                                          ; error checking (DERRCK).
                                          ;
C061  A9 01              LDA   #1         ; logical file 1
C063  A2 08              LDX   #8         ; device number of disk drive
C065  A0 00              LDY   #0         ; secondary address of zero causes relative
                                          ; LOAD
C067  20 BA FF           JSR   SETLFS     ; 1,8,0 is set for relative LOAD
                                          ; include the following three instructions on
                                          ; the 128.
                                          ; LDA BNKNUM; bank number for data
                                          ; LDX BNKFNM; bank containing the ASCII
                                          ; filename
                                          ; JSR SETBNK
C06A  A9 0E              LDA   #FNLENG    ; length of filename
C06C  A2 7D              LDX   #<FILENM   ; address of filename
C06E  A0 C0              LDY   #>FILENM
C070  20 BD FF           JSR   SETNAM     ; set up filename
C073  A9 00              LDA   #0         ; flag for load
C075  A2 00              LDX   #<LOADAD   ; set the load address
C077  A0 40              LDY   #>LOADAD
C079  20 D5 FF           JSR   LOAD       ; load the file at LOADAD
                                          ;
                                          ; JSR DERRCK; insert here for disk error
                                          ; checking
                                          ;
```

```
C07C  60                    RTS
                                                      ;
                                                      ; Insert DERRCK here if you're including
                                                      ; error checking.
                                                      ;
C07D  30  3A  53  FILENM    .ASC   "0:SCREEN CODES"
                                                      ; name of file stored in form of screen codes
C08B              FNLENG    =      * - FILENM         ; length of filename
                                                      ; Include the next two variables on the 128:
                                                      ; BNKNUM .BYTE 0; bank number where
                                                      ; program is to be loaded
                                                      ; BNKFNM .BYTE 0; bank number where
                                                      ; ASCII filename is located
                                                      ;
C08B  00  00      EOF       .WORD0                    ; two-byte end-of-buffer pointer
C08D  00          MAXIMY    .BYTE0                    ; low byte counter for buffer
C08E  00          TEMPY     .BYTE0                    ; temporary storage for .Y
```

See also CASSCR, CASTAS, CNVERT, TASCAS.

Name

Scroll down a line with INST character

Description

This is the first of several scroll-down routines. The technique of scrolling lines from top to bottom is most often used in games where you need to have bombs dropping from the sky (action in space), trees falling toward you (skiing/dodging action), or road signs/highways moving toward you (automobile action). The basic idea is that the player resides at the bottom of the screen, and things are scrolling toward the hapless hero.

Prototype

1. Unlink the first and second screen lines.
2. Get to the top left corner by printing a {HOME} character.
3. Print {DOWN} to move the cursor to line 2.
4. Back up with {LEFT}.
5. Print the {INST} character, which opens up a line.

Explanation

On the 64, the width of a *physical* screen line is 40 characters. A *logical* line, on the other hand, can contain up to 80 characters. A logical line may thus consist of one or two physical lines. A table that starts at location 217 indicates whether a specific physical line is linked to the previous line as part of a single logical line. If the high bit of a lines entry in the table is zero, the line in question is connected to the previous line.

This program puts the cursor in the top left corner, moves down to the second line, backs up, and inserts a character. If the top logical line is fewer than 40 characters long, the technique works; it opens up a second physical line. If the logical line at the top of the screen consists of two physical lines, the technique won't work. So we make sure the the top two lines are unlinked by ORing location 218 with 128 at the start of the routine. The rest is just loading ASCII characters and printing them.

Routine

```
C000            LDTB1    =    217
C000            CHROUT   =    $FFD2
```

SCRDN1 (64 only)

```
C000  A5 DA      SCRDN1  LDA  LDTB1+1     ; entry for second screen line
C002  09 80              ORA  #%10000000  ; undo the line link
C004  85 DA              STA  LDTB1+1
C006  A9 13              LDA  #19         ; HOME character
C008  20 D2 FF           JSR  CHROUT
C00B  A9 11              LDA  #17         ; CURSOR DOWN character
C00D  20 D2 FF           JSR  CHROUT
C010  A9 9D              LDA  #157        ; CURSOR LEFT—to end of first line
C012  20 D2 FF           JSR  CHROUT
C015  A9 94              LDA  #148        ; INSERT character
C017  20 D2 FF           JSR  CHROUT
                                          ; Now lines 2-25 have scrolled down.
C01A  60                 RTS
```

See also BIGMAP, SCRDN2, SCRDN3.

Name

Scroll the screen down a line with the ROM insert routine

Description

A built-in BASIC ROM routine (on the 64) inserts a line and, at the same time, scrolls the lines below it down one notch. By calling this routine, you can cause the whole screen (except the top line) to scroll down.

Prototype

1. Unlink the top line from the second line.
2. Print the {HOME} character to get to the top left corner.
3. Call the ROM routine that inserts a line.

Explanation

BASIC needs the INSLINE routine when a programmer happens to type beyond the fortieth character on a line (see **SCRDN1** for a fuller explanation of physical lines and logical lines). So, if we can unlink the two lines and put the cursor in place, it's quite easy to call the ROM routine that opens up a line.

Note: For the same effect on the 128, you may use the ESC-I sequence to insert a blank line or the ESC-W sequence to scroll the whole screen down by one line.

Routine

```
C000              LDTB1    =    217
C000              CHROUT   =    $FFD2
C000              INSLIN   =    $E965         ; ROM routine to insert a line
                                              ;
C000  A5 DA       SCRDN2   LDA  LDTB1+1       ; entry for second screen line
C002  09 80                ORA  #%10000000    ; undo the line link
C004  85 DA                STA  LDTB1+1
C006  A9 13                LDA  #19           ; HOME character
C008  20 D2 FF             JSR  CHROUT
C00B  20 65 E9             JSR  INSLIN
C00E  60                   RTS
```

See also BIGMAP, SCRDN1, SCRDN3.

Name

Scroll down a line of the screen by copying screen and color memory

Description

This is one of three scroll down routines, and it's by far the longest. The other two routines depend on built-in ROM routines, while this is a stand-alone program that by itself copies characters (and color memory) byte by byte.

Prototype

1. Set up a zero-page pointer to the second-to-the-last screen line.
2. Set up a pointer to the last screen line.
3. Copy 40 characters (and 40 color bytes) from one line to the next.
4. Subtract 40 from each pointer.
5. Continue the loop until 24 lines have been copied.
6. Clear the first line with spaces.

Explanation

The key to this routine is using zero-page pointers. The FROM and the TO pointers tell the subroutines where to copy from and where to put the result. The most important subroutine is COPYFT ($C040), which does four things: It copies 40 characters of screen memory (FROM to TO), changes the pointers so they point to screen memory, copies 40 bytes of color memory (FROM to TO again), and changes the pointers so they point back to the screen.

The FRTOTO subroutine is very general. It copies 40 bytes from the pointer at FROM to the pointer at TO. Because it's generic, it can be used for copying both screen memory and color memory.

The main program initially sets up the pointer at FROM ($C000–$C013) and then calls FROMTO, which creates the second pointer at TO. The X register starts at 24 and counts down to zero because 24 lines must be copied.

You'll see the heart of the program at BIGLOP ($C01A–$C022). JSR to the copy routine (COPYFT), which copies a line down. Next, JSR to MINUS40, which backs up the pointers to the previous line. Then, DEX and BNE to complete the loop.

The final task is to fill the top line with blank spaces (screen code 32) by storing directly to screen memory.

Routine

```
C000                FROM     =    $FB          ; pointer to copy from
C000                TO       =    $FD          ; copy to this area
C000                SCREEN   =    1024         ; screen memory base address
C000                COLOR    =    55296        ; color memory base address
C000                OFFSET   =    COLOR-SCREEN
                                               ; the difference
                                               ;
C000  A9 00         SCRDN3   LDA  #<SCREEN     ; low byte of screen address
C002  85 FB                  STA  FROM
C004  A9 04                  LDA  #>SCREEN     ; high byte of screen address
C006  85 FC                  STA  FROM+1
                                               ; FROM now points to the screen,
                                               ; but we're scrolling down, so we have to
                                               ; adjust by adding 23 lines of 40.
C008  A9 98                  LDA  #<920        ; 23 times 40
C00A  18                     CLC
C00B  65 FB                  ADC  FROM
C00D  85 FB                  STA  FROM
C00F  A9 03                  LDA  #>920
C011  65 FC                  ADC  FROM+1
C013  85 FC                  STA  FROM+1
                                               ; FROM is set up—points to second-to-the-
                                               ; last line.
C015  20 2E C0               JSR  FROMTO       ; subroutine to add 40 to FROM
C018  A2 18                  LDX  #24          ; number of lines to copy
                                               ;
C01A  20 4D C0      BIGLOP   JSR  COPYFT       ; copy a line (screen and color)
C01D  20 3C C0               JSR  MINUS40      ; back up a line
C020  CA                     DEX
C021  D0 F7                  BNE  BIGLOP
                                               ;
                                               ; The lines are copied.
                                               ; Now clear the first line.
C023  A0 27                  LDY  #39
C025  A9 20                  LDA  #32
C027  99 00 04      CLLN     STA  SCREEN,Y
C02A  88                     DEY
C02B  10 FA                  BPL  CLLN
C02D  60                     RTS
                                               ; Subroutines
C02E  A5 FB         FROMTO   LDA  FROM         ; add 40 to FROM pointer
C030  18                     CLC
C031  69 28                  ADC  #40
C033  85 FD                  STA  TO
C035  A5 FC                  LDA  FROM+1
C037  69 00                  ADC  #0           ; add zero in case of a carry
C039  85 FE                  STA  TO+1
C03B  60                     RTS
                                               ;
C03C  A5 FB         MINUS40  LDA  FROM         ; this subroutine subtracts 40
C03E  38                     SEC
C03F  E9 28                  SBC  #40          ; down by 40
C041  85 FB                  STA  FROM
C043  A5 FC                  LDA  FROM+1
C045  E9 00                  SBC  #0           ; subtract zero to adjust for wraparound
C047  85 FC                  STA  FROM+1
C049  20 2E C0               JSR  FROMTO       ; now adjust TO pointer
C04C  60                     RTS
                                               ;
                                               ; Now copy screen and color memory.
C04D  20 5A C0      COPYFT   JSR  FRTOTO       ; copy from FROM to TO
C050  20 64 C0               JSR  FIXCLR       ; change to color memory
```

```
C053  20  5A  C0            JSR   FRTOTO       ; copy color memory from FROM to TO
C056  20  75  C0            JSR   FIXSCN       ; change back to screen
C059  60                    RTS
                                               ;
C05A  A0  27      FRTOTO    LDY   #39          ; get ready to copy 40 bytes (0-39)
C05C  B1  FB      FTTLOP    LDA   (FROM),Y
C05E  91  FD                STA   (TO),Y
C060  88                    DEY                ; count down
C061  10  F9                BPL   FTTLOP       ; branch on plus because we want #0
C063  60                    RTS
                                               ;
C064  A5  FB      FIXCLR    LDA   FROM         ; add offset to FROM and TO
C066  18                    CLC
C067  69  00                ADC   #<OFFSET
C069  85  FB                STA   FROM
C06B  A5  FC                LDA   FROM + 1
C06D  69  D4                ADC   #>OFFSET
C06F  85  FC                STA   FROM + 1
C071  20  2E  C0            JSR   FROMTO       ; add 40 to adjust TO
C074  60                    RTS
                                               ;
C075  A5  FB      FIXSCN    LDA   FROM         ; fix color back to screen memory
C077  38                    SEC
C078  E9  00                SBC   #<OFFSET
C07A  85  FB                STA   FROM
C07C  A5  FC                LDA   FROM+1
C07E  E9  D4                SBC   #>OFFSET
C080  85  FC                STA   FROM + 1
C082  20  2E  C0            JSR   FROMTO       ; not really necessary
C085  60                    RTS
```

See also BIGMAP, SCRDN1, SCRDN2.

Name

Scratch (erase) a disk file

Description

This routine erases a disk file using the DOS scratch command.

Prototype

1. Open the command channel to the drive (SETLFS, SETNAM, OPEN).
2. As part of the SETNAM routine, send the scratch command.
3. Close the file.

Explanation

The first three lines set up the .A, .X, and .Y registers for the call to SETLFS. Before calling SETNAM, we have to put the length of the filename into .A and a pointer to the filename into .X and .Y. But when the command channel (15) is being opened, the filename is really a DOS command. When the Kernal OPEN routine is called, the scratch information is sent to the disk drive. All that remains is the channel closing.

Routine

```
C000                SETLFS   =     $FFBA
C000                SETNAM   =     $FFBD
C000                OPEN     =     $FFC0
C000                CLOSE    =     $FFC3
C000                CLRCHN   =     $FFCC
                                                  ;
C000  A9 01         SCRTCH   LDA   #1             ; logical file number
C002  A2 08                  LDX   #8             ; device number for disk drive
C004  A0 0F                  LDY   #15            ; command channel 15
C006  20 BA FF               JSR   SETLFS         ; prepare to open it
C009  A9 0C                  LDA   #BUFLEN        ; length of buffer
C00B  A2 1E                  LDX   #<BUFFER       ; .X and .Y hold the
C00D  A0 C0                  LDY   #>BUFFER       ; address of the buffer
C00F  20 BD FF               JSR   SETNAM         ; set name
C012  20 C0 FF               JSR   OPEN           ; open it
C015  A9 01                  LDA   #1             ; and immediately
C017  20 C3 FF               JSR   CLOSE          ; close the command channel
C01A  20 CC FF               JSR   CLRCHN         ; clear the channels
C01D  60                     RTS                  ; all done
                                                  ; data area
C01F  53 30 3A      BUFFER   .ASC  "S0:FILENAME"
                                                  ; replace FILENAME with the name of the
                                                  ; file to be scratched
C029  0D                     .BYTE 13             ; return character
C02A                BUFLEN   =     * - BUFFER
```

See also CONCAT, COPYFL, FORMAT, INITLZ, RENAME, VALIDT.

SHFCHK

Name

Check the status of the shift keys

Description

The shift key flag (SHFLAG) at location 653 (location 211 on the 128) can be checked to see whether the SHIFT, Commodore, or CTRL keys are being pressed. On the 128, SHFLAG can also tell you whether the ALT or CAPS LOCK keys are being pressed.

Pressing SHIFT returns a value of 1 in SHFLAG; pressing the Commodore key returns a 2; and pressing CTRL, a 4. On the 128, ALT returns an 8; CAPS LOCK, a 16. If two or more of these keys are pressed simultaneously, SHFLAG returns the sum of these values. For example, pressing CTRL and SHIFT together result in a value of 5 in SHFLAG.

Prototype

Return the contents of the SHIFT flag in .A.

Explanation

In the example routine, the current contents of SHFLAG are continually printed on the screen. Press the SHIFT, Commodore, and CTRL keys (also the ALT and CAPS LOCK keys on the 128), either alone or together, to see the effect on SHFLAG. Press Q to exit (quit) the routine.

Routine

```
C000                SHFLAG   =     653        ; SHFLAG = 211 on the 128—shift key flag
C000                CHROUT   =     65490
C000                GETIN    =     65508
C000                CLRHOM   =     58692      ; CLRHOM = 49474 on the 128
C000                LINPRT   =     48589      ; LINPRT = 36402 on the 128
                                              ;
                                              ; Check shift flag. Print result. Quit when Q
                                              ; is pressed.
C000  20 44 E5  CLRROM   JSR   CLRHOM     ; clear screen
C003  20 19 C0  LOOP     JSR   SHFCHK     ; check shift flag
C006  AA                 TAX              ; use flag value as low byte
C007  A9 00              LDA   #0         ; zero in the high byte
C009  20 CD BD           JSR   LINPRT     ; print a two-byte integer to screen (see
                                          ; NUMOUT)
C00C  A9 0D              LDA   #13        ; print RETURN
C00E  20 D2 FF           JSR   CHROUT
C011  20 E4 FF           JSR   GETIN      ; get a key
C014  C9 51              CMP   #81        ; is it Q?
C016  D0 EB              BNE   LOOP       ; no, so LOOP
C018  60                 RTS              ; yes, so return
                                          ;
                                          ; Return SHFLAG in .A.
C019  AD 8D 02  SHFCHK   LDA   SHFLAG
C01C  60                 RTS
```

See also STPFLG, STPKER.

Name

Clear the SID chip

Description

SIDCLR stores a zero in each of the SID chip's 25 write-only registers, thereby cancelling all sound output.

Prototype

In a loop, store zeros in memory in the range 54272–54296 and RTS.

Explanation

Generally, the first step you take in writing any sound routine is to clear the SID chip so that parameters remaining from a previous use of the chip won't affect the current sound.

A minor problem with the SID chip is that it sometimes continues to echo the last frequency output even after the intended sound has finished. This effect, though barely audible, may annoy the user. If it does, you can silence the chip altogether by JSRing to **SIDCLR** at the end of your sound routines.

Note: The SID chip is addressed at locations 54272–54300, a total of 29 registers. The first 25, cleared in **SIDCLR**, are *write-only registers,* meaning they can't be read. In contrast, the remaining 4 are *read-only registers;* writing to them has no effect.

Routine

```
C000               FRELO1   =     54272      ; starting address of the SID chip
                                             ;
C000  A9 00        SIDCLR   LDA   #0         ; fill with zeros
C002  A0 18                 LDY   #24        ; as the offset from FRELO1
C004  99 00  D4    SIDLOP   STA   FRELO1,Y   ; store zero in each SID register
C007  88                    DEY              ; for next lower address
C008  10 FA                 BPL   SIDLOP     ; fill 25 bytes
C00A  60                    RTS              ; we're done
```

See also BEEPER, BELLRG, EXPLOD, INTMUS, MELODY, NOTETB, SIDVOL, SIRENS.

Name

Set the SID chip volume register

Description

SIDCLR sets the SID chip volume register to the level (0–15) specified in the accumulator.

Prototype

1. Enter this routine with the chosen volume level in the accumulator.
2. Store this value into the volume and filter-select register at 54296 (SIGVOL) and return to the calling program.

Explanation

SIGVOL (location 54296) is a multifaceted, write-only register for the SID chip. With it, you can choose the volume of the sound that's output (bits 0–3), select filtering (bits 4–6), or disconnect the output of voice 3. In this routine, the register's sole purpose is to determine the volume level for the chip. The range of the volume level is 0 (minimum) through 15 (maximum).

SIDVOL is easy to use. Just load a number representing the volume into the accumulator and JSR to the routine. In the example, we set the chip to its maximum volume level of 15.

Note: Some programmers attempt to silence the SID chip by storing a zero in 54296, but this is not always effective. A better approach is to store a zero in the frequency registers or turn the chip off completely with **SIDCLR**.

Routine

```
C000                SIGVOL  =    54296     . volume and filter-select register
                                           ;
                                           ; Set the volume to 15.
C000  A9  0F        MAIN    LDA  #15       ; load A with the volume, 0 (minimum)
                                           ; through 15 (maximum)
C002  4C  05  C0            JMP  SIDVOL    ; set the volume to A
                                           ;
                                           ; Enter with the volume in .A.
C005  8D  18  D4    SIDVOL  STA  SIGVOL    ; store the volume value in .A into the
                                           ; volume register
C008  60                    RTS
```

See also BEEPER, BELLRG, EXPLOD, INTMUS, MELODY, NOTETB, SIDCLR, SIRENS.

Name

Produce a siren sound

Description

SIRENS causes the SID chip to emit an extended sirenlike sound. At certain intervals in a game, you could use it to signal to the user that he's reached a higher level or achieved bonus points. Or you could use it as fanfare at the conclusion of the game.

Prototype

1. Clear the SID chip with **SIDCLR**.
2. Set up the necessary SID chip parameters for voice 1. Set sustain/release to $F0, select a sawtooth waveform, and gate the sound.
3. Assign a low frequency and a triangle waveform to voice 3.
4. Disconnect output from voice 3. At the same time, select band-pass filtering and the volume.
5. Store %00000001 in the filter/resonance control register to filter voice 1 without resonance.
6. Select a band-pass filter cutoff frequency.
7. In SIRLOP, multiply the output of voice 3 by 32 and add in a base frequency of 15000. Store the low and high bytes of the resulting frequency in voice 1.
8. Pause four jiffies before getting another frequency value for voice 3.
9. Repeat SIRLOP 256 times. Then clear the chip and RTS.

Explanation

In this routine, the output from voice 3 modulates the frequency of voice 1. In the process, voice 3 is not actually heard. As a result, no SID attack/decay or sustain/release parameters are required for this voice. Its only use is in providing a frequency value for voice 1.

After disconnecting the audio output of voice 3, the waveform (high byte only) for this voice is read from RANDOM. Since a triangle waveform is selected for voice 3, the numbers returned by RANDOM increase gradually from 0 to 255, and then work down to 0 again. In order to get a suitable frequency range for voice 1, these values are multiplied by 32 and then added to a base frequency of 15000.

Another feature of **SIRENS** is its use of band-pass filtering. With the band-pass filter implemented, frequencies on either side of a cutoff frequency are diminished in volume.

Since only 11 bits on the two-byte cutoff register are addressed, the cutoff filter value can range from 0–2047. Although the number stored in this register is proportional to the cutoff frequency (in this case, 616), the value itself does not represent an actual frequency. Probably the best way to achieve the effect you're looking for with this register is through experimentation.

Routine

```
C000                ZP       =    251
C000                JIFFLO   =    162          ; low byte of jiffy clock
C000                FRELO1   =    54272        ; voice 1 frequency control (low byte)
C000                FREHI1   =    54273        ; voice 1 frequency control (high byte)
C000                VCREG1   =    54276        ; voice 1 control register
C000                SUREL1   =    54278        ; voice 1 sustain/release register
C000                FRELO3   =    54286        ; voice 3 frequency control (low byte)
C000                VCREG3   =    54290        ; voice 3 control register
C000                CUTLO    =    54293        ; lower three bits of filter cutoff frequency
C000                CUTHI    =    54294        ; filter cutoff frequency (high byte)
C000                RESON    =    54295        ; filter/resonance control register
C000                SIGVOL   =    54296        ; volume and filter select register
C000                RANDOM   =    54299        ; reads high byte of voice 3
C000                BASFRE   =    15000        ; base frequency to add to voice 3
                                               ;
C000  20 64 C0  SIRENS  JSR  SIDCLR            ; go clear the SID chip
C003  A9 F0             LDA  #$F0              ; set full sustain/fastest release
C005  8D 06 D4          STA  SUREL1
C008  A9 21             LDA  #%00100001        ; select sawtooth waveform (voice 1) and
                                               ; gate sound
C00A  8D 04 D4          STA  VCREG1
C00D  A9 02             LDA  #2                ; give voice 3 a frequency
C00F  8D 0E D4          STA  FRELO3
C012  A9 10             LDA  #%00010000        ; select triangle waveform (voice 3)
C014  8D 12 D4          STA  VCREG3
C017  A9 AF             LDA  #%10101111        ; disconnect voice 3 output/select band-
                                               ; pass/max. volume
C019  8D 18 D4          STA  SIGVOL
C01C  A9 01             LDA  #%00000001        ; no resonance and filter voice 1
C01E  8D 17 D4          STA  RESON
C021  A9 00             LDA  #0                ; select band-pass cutoff frequency of 616
C023  8D 15 D4          STA  CUTLO
C026  A9 4D             LDA  #77
C028  8D 16 D4          STA  CUTHI
C02B  A2 00             LDX  #0                ; as an index in SIRLOP
                                               ; Calculate voice 1 frequency from voice 3
                                               ; frequency (high byte).
C02D  A9 00     SIRLOP  LDA  #0                ; initialize voice 1 frequency (high byte)
C02F  85 FC             STA  ZP+1
C031  AD 1B D4          LDA  RANDOM            ; get voice 3 frequency (high byte)
C034  85 FB             STA  ZP                ; store in zero page as low byte
C036  06 FB             ASL  ZP                ; multiply it by 32, double low byte
C038  26 FC             ROL  ZP+1              ; then high byte
C03A  06 FB             ASL  ZP                ; double four more times
C03C  26 FC             ROL  ZP+1
C03E  06 FB             ASL  ZP
C040  26 FC             ROL  ZP+1
C042  06 FB             ASL  ZP
C044  26 FC             ROL  ZP+1
C046  06 FB             ASL  ZP
```

```
C048  26  FC                 ROL   ZP+1
                                               ; Add a base frequency of 15000 to this.
C04A  A5  FB                 LDA   ZP          ; low byte first
C04C  18                     CLC               ; for addition
C04D  69  98                 ADC   #<BASFRE    ; add low byte of base frequency
C04F  8D  00  D4             STA   FRELO1      ; and store in voice 1 frequency register
                                               ; (low byte)
C052  A5  FC                 LDA   ZP+1        ; then high byte
C054  69  3A                 ADC   #>BASFRE    ; add high byte of base frequency
C056  8D  01  D4             STA   FREHI1      ; and store in voice 1 frequency register
                                               ; Delay four jiffies.
C059  A9  04                 LDA   #4          ; add four jiffies to jiffy clock reading
C05B  65  A2                 ADC   JIFFLO
C05D  C5  A2     DELAY       CMP   JIFFLO      ; and wait for four jiffies to elapse
C05F  D0  FC                 BNE   DELAY
C061  CA                     DEX               ; for next note
C062  D0  C9                 BNE   SIRLOP      ; repeat SIRLOP 256 times
                                               ;
                                               ; Fall through to SIDCLR to stop sound and
                                               ; RTS
                                               ; Clear the SID chip.
C064  A9  00     SIDCLR      LDA   #0          ; fill with zeros
C066  A0  18                 LDY   #24         ; as the offset from FRELO1
C068  99  00  D4 SIDLOP      STA   FRELO1,Y    ; store 0 in each SID chip address
C06B  88                     DEY               ; for next lower address
C06C  10  FA                 BPL   SIDLOP      ; fill 25 bytes
C06E  60                     RTS               ; we're done
```

See also BEEPER, BELLRG, EXPLOD, INTMUS, MELODY, NOTETB, SIDCLR, SIDVOL.

Name

Sprite interrupt routine—automatic sprite movement

Description

In a situation where you need sprites to travel automatically from one spot to another, this routine may be helpful. It makes a sprite operate like a battery-powered toy car. Turn it on, and it moves forward without any further attention from you. The sprite position is updated 60 times a second, regardless of what the main program is doing.

Prototype

First, install the routine:

1. Create the sprite shape and set up the necessary pointers.
2. Set the interrupt-disable flag (SEI).
3. Change the interrupt vector to point to the **SPRINT** routine.
4. Clear the interrupt flag (CLI) and return.

Within the routine:

5. Slow down the movement by checking a flag (if necessary).
6. Change the sprite shape (optional).
7. Update the *x* and *y* positions, and store them in registers.
8. Jump to the normal interrupt-handling routine.

Explanation

In machine language, sprite movement can be something of a headache. One problem is that ML is very fast; a sprite-mover routine can easily move a single sprite from one edge of the screen to another in the blink of an eye. A delay loop is an unsatisfactory solution—you want the sprites to slow down, not the whole program. A second problem is that updating sprite positions can take a large number of instructions that clutter up the main loop within a program.

Putting the sprites on the interrupt is a workable answer to both difficulties. Every 1/60 second, the wedge takes over and handles automatic sprite movement.

The code at locations \$C000–\$C00A copies two sprite shapes from the program down to the cassette buffer (to put them in the realm of the VIC chip's default 16K video memory bank). Then the code at locations \$C00B–\$C026 sets up the initial *x* and *y* positions, sets the sprite color to white, turns on the sprite, and sets the sprite pointer.

Next, the wedge is installed. It's necessary to use the SEI instruction to disable interrupts while the installation is in the

works. Otherwise, an interrupt may occur during the change, and the 6510/8502 may jump to an unusual location in memory. The IRQ vector at locations 788–779 is changed to point to **SPRINT**. Henceforth, all IRQ interrupts will move to our own routine before continuing to the normal interrupt-handling routine. When the wedge is complete, CLI clears the flag, and RTS returns the program to BASIC (or to the ML routine that called it).

The **SPRINT** routine is now called 60 times a second. Only one time in four does it actually do something (15 times a second is plenty fast). This portion of the program could be eliminated or modified.

First, at $C03A, **SPRINT** checks the current shape of the sprite. If it's S1, the shape is changed to S2 and vice versa. You're not required to change the shapes; this section could also be eliminated. Next, at $C04E, the *x* and *y* positions are updated. In this example, the *x* position is increased by two, and one is added to the *y* position (this could be changed, depending on the program). The *x* and *y* positions are variables stored in memory. After they're changed, they must be copied to the appropriate sprite registers (at $C068–$C079).

The routine finishes up, not with an RTS, but with a JMP to the normal IRQ handler (NORIRQ, $EA31 on the 64). This routine scans the keyboard and generally keeps things running.

Note: The XHI variable is copied directly to SPXM because only sprite 0 is being moved. If you use sprites 1–7, it will be necessary to shift the bits to the left to put the high bit in the correct position.

On the 128, you must disable the sprite control commands of BASIC. Before SYSing to this routine, enter POKE 4861,1 (or use any other non-zero value). Alternately, you could LDA and STA at the start of the program.

Warning: It's important not to overload the interrupt routine with too many instructions. The interrupt handler is called every 1/60 second, which seems very fast to us. But to the computer, which works in millionths of a second, it's a long time. If you write an extremely long interrupt wedge, it may possibly require more than 1/60 second to run. If this happens, the interrupt routine will run in the background, and, by the time it's done, another interrupt will have occurred. The main program will never have a chance to execute.

Routine

```
C000                JIF     =   $A2           ; lowest byte of the jiffy clock
C000                SPR1    =   $0340         ; SPR1 = $0E00 on the 128
C000                SPR2    =   $0380         ; SPR2 = $0E40 on the 128
C000                S1      =   13            ; S1 = 56 on the 128—pointer 1 to $0340
C000                S2      =   14            ; S2 = 57 on the 128—pointer 2 to $0380
C000                SPCOLR  =   53287         ; sprite 0 color
C000                SPX     =   53248         ; x position
C000                SPY     =   53249         ; y position
C000                SPXM    =   53264         ; MSB bit of x position
C000                SPE     =   53269         ; sprite enable
C000                SPP     =   2040          ; pointer to sprite 0
C000                IRQVEC  =   788
C000                NORIRQ  =   $EA31         ; NORIRQ = $FA65 on the 128—normal IRQ
                                              ; handling routine
                                              ;
C000  A2 80                 LDX #128          ; two sprites = 128 bytes
C002  BD 80 C0      COPY    LDA SHAPE1,X      ; copy from the program
C005  9D 40 03              STA SPR1,X        ; to available memory
C008  CA                    DEX
C009  10 F7                 BPL COPY          ; cutting it thin (127 is plus, 128 is minus)
                                              ;
C00B  A9 64                 LDA #100
C00D  8D 7D C0              STA XLO           ; put it in x-position shadow
C010  8D 7F C0              STA YLO           ; and y-position shadow
C013  A9 01                 LDA #1            ; the color white
C015  8D 27 D0              STA SPCOLR        ; into the color register
C018  A9 00                 LDA #0            ; no MSB
C01A  8D 7E C0              STA XHI           ; into the shadow register
C01D  A9 01                 LDA #1            ; enable sprite 0
C01F  8D 15 D0              STA SPE
C022  A9 0D                 LDA #S1           ; sprite shape 1 pointer
C024  8D F8 07              STA SPP           ; into 2040
                                              ;
                                              ; change the IRQ vector now
C027  78                    SEI               ; first stop interrupts
C028  A9 34                 LDA #<SPRINT      ; change the vector
C02A  8D 14 03              STA IRQVEC
C02D  A9 C0                 LDA #>SPRINT
C02F  8D 15 03              STA IRQVEC+1
C032  58                    CLI               ; clear the interrupts
C033  60                    RTS               ; and we're done with setup
                                              ;
C034                SPRINT  =   *             ; this is the interrupt routine
C034  A5 A2                 LDA JIF           ; every fourth interrupt
C036  29 03                 AND #%00000011    ; AND it with 3
C038  D0 40                 BNE ENSPRIN       ; if a bit's on, quit
C03A  AD F8 07              LDA SPP           ; get the pointer
C03D  C9 0D                 CMP #S1           ; is it shape 1?
C03F  D0 08                 BNE DO1           ; no, do shape 1
C041  A9 0E                 LDA #S2           ; load shape 2
C043  8D F8 07              STA SPP           ; and store it
C046  4C 4E C0              JMP XY            ; go ahead to x and y
C049  A9 0D         DO1     LDA #S1           ; get shape 1
C04B  8D F8 07              STA SPP           ; and store the pointer
C04E  AD 7D C0      XY      LDA XLO           ; find the low byte (XLO)
C051  18                    CLC
C052  69 02                 ADC #2            ; add two
C054  8D 7D C0              STA XLO           ; and store it back
C057  AD 7E C0              LDA XHI           ; check the high byte
C05A  69 00                 ADC #0            ; add zero or one
C05C  C9 02                 CMP #2            ; if it's not two
```

```
C05E  D0 02                BNE   STHI          ; branch ahead
C060  A9 00                LDA   #0            ; otherwise, make it zero
C062  8D 7E  C0  STHI      STA   XHI
                                               ;
C065  EE 7F  C0            INC   YLO           ; add one to the y position
                                               ; Now change the positions.
C068  AD 7E  C0            LDA   XHI
C06B  8D 10  D0            STA   SPXM
C06E  AD 7D  C0            LDA   XLO
C071  8D 00  D0            STA   SPX
C074  AD 7F  C0            LDA   YLO
C077  8D 01  D0            STA   SPY
C07A  4C 31  EA  ENSPRIN   JMP   NORIRQ        ; do the normal IRQ stuff
                                               ;
C07D  00         XLO       .BYTE 0
C07E  00         XHI       .BYTE 0
C07F  00         YLO       .BYTE 0
C080             SHAPE1    =     *
C080  35 00 00             .BYTE %00110101,%0000000,%00000000
C083  1A 7C 00             .BYTE %00011010,%1111100,%00000000
C086  0D 66 00             .BYTE %00001101,%1100110,%00000000
C089  46 42 06             .BYTE %01000110,%1000010,%00000110
C08C  21 42 1E             .BYTE %00100001,%1000010,%00011110
C08F  08 24 70             .BYTE %00001000,%0100100,%01110000
C092  07 5B C0             .BYTE %00000111,%1011011,%11000000
C095  06 3F 00             .BYTE %00000110,%0111111,%00000000
C098  0E 3C 00             .BYTE %00001110,%0111100,%00000000
C09B  06 3C 00             .BYTE %00000110,%0111100,%00000000
C09E  01 7C 00             .BYTE %00000001,%1111100,%00000000
C0A1  01 7C 00             .BYTE %00000001,%1111100,%00000000
C0A4  00 7C 00             .BYTE %00000000,%1111100,%00000000
C0A7  E1 7F 00             .BYTE %11100001,%1111111,%00000000
C0AA  E1 78 80             .BYTE %11100001,%1111000,%10000000
C0AD  62 20 C0             .BYTE %01100010,%0100000,%11000000
C0B0  2E 40 60             .BYTE %00101110,%1000000,%01100000
C0B3  1D 40 70             .BYTE %00011101,%1000000,%01110000
C0B6  00 03 C0             .BYTE %00000000,%0000011,%11000000
C0B9  00 07 80             .BYTE %00000000,%0000111,%10000000
C0BC  00 02 C0             .BYTE %00000000,%0000010,%11000000
C0BF  00                   .BYTE 0                     ; zero to make it even
C0C0             SHAPE2    =     *
C0C0  00 00 00             .BYTE %00000000,%0000000 %00000000
C0C3  3B 78 00             .BYTE %00111011,%1111000,%00000000
C0C6  2B 4C 00             .BYTE %00101011,%1001100,%00000000
C0C9  46 44 00             .BYTE %01000110,%1000100 %00000000
C0CC  21 42 00             .BYTE %00100001,%1000010,%00000000
C0CF  00 24 70             .BYTE %00000000,%0100100,%01110000
C0D2  01 5B F8             .BYTE %00000001,%1011011,%11111000
C0D5  02 3F B0             .BYTE %00000010,%0111111,%10110000
C0D8  06 38 00             .BYTE %00000110,%0111000 %00000000
C0DB  02 3C C0             .BYTE %00000010,%0111100,%11000000
C0DE  01 7D E0             .BYTE %00000001 %1111101,%11100000
C0E1  01 7C 00             .BYTE %00000001,%1111100,%00000000
C0E4  00 7C 00             .BYTE %00000000,%1111100 %00000000
C0E7  01 7F 00             .BYTE %00000001,%1111111,%00000000
C0EA  01 78 80             .BYTE %00000001,%1111000,%10000000
C0ED  00 23 C0             .BYTE %00000000,%0100011,%11000000
C0F0  03 73 80             .BYTE %00000011,%1110011,%10000000
C0F3  18 7B C0             .BYTE %00011000,%1111011,%11000000
C0F6  70 3F 80             .BYTE %01110000,%0111111,%10000000
C0F9  00 3D 80             .BYTE %00000000,%0111101,%10000000
C0FC  0C 02 C0             .BYTE %00001100,%0000010,%11000000
```

See also MOVSAB.

Name

Calculate the integer square root of an integer value

Description

Because squares follow a definite pattern, it's fairly easy to find the integer square root of a given number. Note that this routine doesn't handle the fractional part of a square root. For example, it will return 3 as the square root for all the numbers in the range 9–15 and ignore the fractional component.

Prototype

1. Store the value of which you want to find the square root in VAL.
2. Initialize ADDBT and SQUARE to one, and ROOT to negative one.
3. Increment ROOT (so it starts as zero).
4. Compare SQUARE to VAL.
5. If SQUARE is equal or larger, exit the routine. The result is in ROOT.
6. If SQUARE is smaller, add 2 to ADDBT.
7. Add ADDBT to SQUARE and loop back to step 3.

Explanation

Normally, finding the square root of a number is a fairly involved process. But if you're working with integers, you may not care about the fractional part of the result. In that case, we can use a mathematical property of squares to find the integer portion of the square root.

Write down the first six squares and underneath write down the first six odd numbers; then add up the columns:

0	1	4	9	16	25
1	3	5	7	9	11
1	4	9	16	25	36

Note the pattern of squares is exactly echoed in the sums underneath. It can be proven mathematically that this sequence continues to infinity. To calculate squares, then it becomes a matter of keeping a counter (ADDBT in the program below) that starts at 1 and increments by 2 during each loop. SQUARE also starts at 1 and has ADDBT added, to yield 4, 9, 16, and so on. The answer, held in ROOT, lags one number behind the actual square root because we want to find a square that's larger than VAL, the number from which we're extracting a root.

The example program prints a bad facsimile of the square-root symbol, and then the number from VAL and an equal sign. The answer is calculated and printed.

Routine

```
C000                LINPRT   =     $BDCD       ; LINPRT = $8E32 on the 128
C000                CHROUT   =     $FFD2
                                               ;
C000  A9 CD                  LDA   #205        ; backslash character
C002  20 D2 FF               JSR   CHROUT      ; print it
C005  A9 CF                  LDA   #207        ; upper left-corner character
C007  20 D2 FF               JSR   CHROUT      ; print it (to make a square-root symbol)
C00A  AE 81 C0               LDX   VAL         ; print the value
C00D  AD 82 C0               LDA   VAL+1       ; high byte
C010  20 CD BD               JSR   LINPRT
C013  A9 3D                  LDA   #61         ; equal sign character
C015  20 D2 FF               JSR   CHROUT      ; print it
C018  20 25 C0               JSR   SQROOT      ; calculate the square root
C01B  AE 87 C0               LDX   ROOT        ; print the square value
C01E  AD 88 C0               LDA   ROOT+1
C021  20 CD BD               JSR   LINPRT
C024  60                     RTS
                                               ;
C025                SQROOT   =     *
C025  A2 01                  LDX   #1          ; start with 1 in ADDBT and SQUARE
C027  8E 85 C0               STX   ADDBT
C02A  8E 83 C0               STX   SQUARE
C02D  CA                     DEX               ; .X = 0, the high byte
C02E  8E 86 C0               STX   ADDBT+1
C031  8E 84 C0               STX   SQUARE+1
C034  CA                     DEX
C035  8E 87 C0               STX   ROOT        ; net result of    1 in ROOT
C038  8E 88 C0               STX   ROOT+1      ; also a 255 into the high byte of ROOT
                                               ;
                                               ; Start by incrementing ROOT.
C03B  EE 87 C0      LOOP     INC   ROOT
C03E  D0 03                  BNE   NOHI
C040  EE 88 C0               INC   ROOT+1
                                               ;
C043                NOHI     =     *           ; Now compare VAL to SQUARE.
C043  AD 82 C0               LDA   VAL+1       ; high byte first
C046  CD 84 C0               CMP   SQUARE+1
C049  F0 03                  BEQ   MAYBE       ; if equal, check low byte
C04B  B0 09                  BCS   MORE        ; if VAL is bigger, do another round
C04D  60            QUIT     RTS               ; else RTS with the result in ROOT
C04E  AD 81 C0      MAYBE    LDA   VAL         ; look at VAL again (low byte)
C051  CD 83 C0               CMP   SQUARE      ; compare it
C054  90 F7                  BCC   QUIT        ; quit if smaller
C056  20 72 C0      MORE     JSR   ADD2
C059  18                     CLC
C05A  AD 83 C0               LDA   SQUARE      ; double add
C05D  6D 85 C0               ADC   ADDBT
C060  8D 83 C0               STA   SQUARE
C063  AD 84 C0               LDA   SQUARE+1
C066  6D 86 C0               ADC   ADDBT+1
C069  8D 84 C0               STA   SQUARE+1
C06C  B0 03                  BCS   ENDIT       ;
C06E  4C 3B C0               JMP   LOOP
```

```
C071  60         ENDIT    RTS
                                          ;
                                          ; Add 2 to ADDBT.
C072  AD 85  C0  ADD2     LDA   ADDBT
C075  18                  CLC
C076  69  02              ADC   #2
C078  8D  85  C0          STA   ADDBT
C07B  90  03              BCC   NOMO
C07D  EE  86  C0          INC   ADDBT+1
C080  [redacted]         NOMO     RTS
                                          ;
C081  C4  32     VAL      .WORD 12996     ; the square of 114
C083  00  00     SQUARE   .BYTE 0,0
C085  00  00     ADDBT    .BYTE 0,0
C087  00  00     ROOT     .BYTE 0,0
```

Name

Binary search of a sorted list

Description

The good news about a binary search like **SRCBIN** is that it's by far the fastest way to find an item in a list. The bad news is that for it to work correctly, the list must already be in alphabetic order. For a static list that doesn't change much—like a dictionary—a binary search is ideal. For a volatile list that changes often, you'll have to spend a significant amount of time keeping it in order.

Prototype

1. Start by setting up pointers to the beginning, the end, and the midpoint of the list.
2. Compare the midpoint to the search string.
3. If it's equal, skip forward to step 5.
4. If the midpoint value is higher than the search string, set the end of the list to the midpoint and calculate a new midpoint. Branch back to step 2.
5. If the midpoint is lower than the sought-for string, set the beginning to the current midpoint and fix the new mid point. Return to step 2.
6. When the search string is found, step backward on the list until the first occurrence is discovered.

Explanation

The *binary* part of a binary search means that the list is divided into two parts. To illustrate how it works, let's first look at how it doesn't work. Imagine that you live in a city that has a phone directory containing 100,000 names, listed in alphabetic order. To find the number for someone named Milt Young, it would be madness for you to start searching at the beginning of the phone book (this is a *sequential* search). You'd have to look at many thousands of names before you found the one you wanted.

For a binary search, you'd open the phone book halfway and check the name there. Let's say it's Meeks. Immediately, you know that the search string (Young) is in the second half of the phone book. With one comparison, you've eliminated half the names on the list. Next, you split the remaining pages in two and check the name. Again, half the names are discarded. Each pass through the loop cuts in half the number of names to be checked. For a list of 256 items, you'd need at

most 8 comparisons to find the target name. For 64K items, you'd need a maximum of 16 comparisons.

The dark side of the binary search is that maintaining the list requires a good deal of effort since it must be in alphabetic or numeric order.

The **SRCBIN** routine is long, but relatively simple to understand. There are three possibilities: The search string is on the list, it's on the list several times, or it's not on the list. If the target is found, the binary search is successful, but just in case there are others, **SRCBIN** moves backward in memory to find the first occurrence. If it's not found, a value of zero is stored into MID. If it is found, a pointer to the first matching string is stored in MID.

The example program first reads an ASCII file into memory (in READFILE) and then alphabetizes it (ALPHA). For a database application, it shouldn't be necessary to alphabetize before the search routine is called. You should keep the list in alphabetic order.

Routine

```
C000                STATUS   =    144
C000                P1       =    $F9
C000                ZP       =    $FB
C000                Z2       =    $FD
C000                LINPRT   =    $BDCD     ; LINPRT = $8E32 on the 128
C000                CHROUT   =    $FFD2
C000                BUFFER   =    $6000     ; where the words are
C000                POINTR   =    $5000     ; where the pointers to the words are
                                            ;
C000  20 29 C1               JSR  READFILE  ; get the words into memory and set up the
                                            ; pointers
C003  20 E4 C1               JSR  ALPHA     ; alphabetize the list
C006  A0 00     GLOOP        LDY  #0
C008  20 CF FF  NLOOP        JSR  CHRIN     ; get a string
C00B  C9 0D                  CMP  #13       ; return
C00D  F0 06                  BEQ  FINDIT    ; done, so look for it
C00F  99 76 C2               STA  SEARCH,Y  ; save it
C012  C8                     INY            ; and count forward
C013  D0 F3                  BNE  NLOOP
C015  A9 00     FINDIT       LDA  #0        ; end it with a zero
C017  99 76 C2               STA  SEARCH,Y
C01A  C0 00                  CPY  #0        ; was there anything (or too much)?
C01C  F0 EA                  BEQ  NLOOP     ; go back
                                            ;
                                            ; Now find the word.
C01E  20 2D C0               JSR  SRCBIN
C021  AE E2 C1               LDX  MID
C024  AD E3 C1               LDA  MID+1
C027  20 CD BD               JSR  LINPRT    ; print the address of the string
C02A  4C 06 C0               JMP  GLOOP
                                            ;
C02D  20 51 C0  SRCBIN       JSR  SETUP     ; set up the TOP, BOT, and MID pointers to
                                            ; the pointers
C030  20 8C C0  SBLOOP       JSR  CHKMID    ; look at MID
```

```
C033  F0 10               BEQ  MOVEDN      ; found it, now back up a little
C035  30 02               BMI  HALF0       ; in the first half
C037  10 06               BPL  HALF1       ; in the second half
                                           ;
C039  20 F0 C0  HALF0     JSR  MIDTOP      ; MID is the new TOP
C03C  4C 30 C0            JMP  SBLOOP      ; go back
C03F  20 F4 C0  HALF1     JSR  MIDBOT      ; MID is the new BOT
C042  4C 30 C0            JMP  SBLOOP      ; and loop
                                           ;
C045  20 05 C1  MOVEDN    JSR  MIDMIN      ; MID minus two
C048  20 BC C0            JSR  CHKMID      ; check it
C04B  F0 F8               BEQ  MOVEDN      ; move down one more
C04D  20 17 C1            JSR  MIDPLS      ; mid plus two
C050  60                  RTS
                                           ;
C051  A2 03     SETUP     LDX  #3
C053  BD DA C1  SET01     LDA  SB,X        ; copy SB and EB
C056  9D DE C1            STA  BOT,X       ; to BOT and TOP
C059  CA                  DEX              ; count down to 255
C05A  10 F7               BPL  SET01       ; loop
                                           ;
C05C  AD E0 C1  MIDSET    LDA  TOP         ; find midpoint
C05F  38                  SEC
C060  ED DE C1            SBC  BOT         ; subtract BOT
C063  29 FC               AND  #%11111100  ; make sure it will be even after the rotate
C065  8D E2 C1            STA  MID         ; store in MID temporarily
C068  AD E1 C1            LDA  TOP+1       ; high byte, too
C06B  ED DF C1            SBC  BOT+1       ; subtracts
C06E  8D E3 C1            STA  MID+1       ; into MID
C071  4E E3 C1            LSR  MID+1       ; cut in half the high
C074  6E E2 C1            ROR  MID         ; and low bytes of MID
                                           ; The halfway point is ready.
C077  AD E2 C1            LDA  MID         ; better check it
C07A  0D E3 C1            ORA  MID+1       ; are any bits on?
C07D  F0 13               BEQ  PANIC       ; no, and we haven't found it
C07F  AD E2 C1            LDA  MID         ; carry is always clear
C082  6D DE C1            ADC  BOT         ; add to BOT
C085  8D E2 C1            STA  MID
C088  AD E3 C1            LDA  MID+1       ; high byte, too
C08B  6D DF C1            ADC  BOT+1
C08E  8D E3 C1            STA  MID+1       ; MID is ready
C091  60                  RTS              ; so we can go back
                                           ;
                                           ; Maybe it's not on the list.
C092  AD DE C1  PANIC     LDA  BOT
C095  8D E2 C1            STA  MID
C098  AD DF C1            LDA  BOT+1
C09B  8D E3 C1            STA  MID+1
C09E  20 BC C0            JSR  CHKMID      ; check it
C0A1  F0 18               BEQ  NOPROB      ; found the string
C0A3  AD E0 C1            LDA  TOP         ; check top
C0A6  8D E2 C1            STA  MID
C0A9  AD E1 C1            LDA  TOP+1
C0AC  8D E3 C1            STA  MID+1
C0AF  F0 0A               BEQ  NOPROB      ; found it
C0B1  68                  PLA              ; get rid of the address
C0B2  68                  PLA              ; from the JSR
C0B3  A9 00               LDA  #0          ; zero out MID
C0B5  8D E2 C1            STA  MID
C0B8  8D E3 C1            STA  MID+1
C0BB  60        NOPROB    RTS
                                           ;
C0BC  AD E2 C1  CHKMID    LDA  MID         ; get the pointer
C0BF  85 FB               STA  ZP          ; to the string
```

```
C0C1  AD E3  C1            LDA   MID+1        ; and store in
C0C4  85  FC               STA   ZP+1         ; ZP
C0C6  A0  01               LDY   #1           ; next,
C0C8  B1  FB               LDA   (ZP),Y       ; the string address
C0CA  85  FE               STA   Z2+1         ; goes into Z2
C0CC  88                   DEY                ; .Y is zero
C0CD  B1  FB               LDA   (ZP),Y
C0CF  85  FD               STA   Z2
                                              ;
                                              ; Compare them.
C0D1  B9  76  C2  CMTHEM   LDA   SEARCH,Y     ; get a character
C0D4  D0  05               BNE   CKM1         ; if not zero, check more
C0D6  11  FD               ORA   (Z2),Y       ; is the string also a zero?
C0D8  D0  10               BNE   TOOHI        ; no, the string is too high
C0DA  60                   RTS                ; else, RTS with the equal flag set
C0DB  AA          CKM1     TAX                ; save it
C0DC  B1  FD               LDA   (Z2),Y
C0DE  F0  0D               BEQ   TOOLOW       ; if Z2 is zero, mid is too low
C0E0  8A                   TXA                ; get .A back
C0E1  D1  FD               CMP   (Z2),Y       ; compare search to Z2, which is MID
C0E3  90  05               BCC   TOOHI        ; MID is too high
C0E5  D0  06               BNE   TOOLOW       ; MID is too low
C0E7  C8                   INY                ; they're equal, so
C0E8  D0  E7               BNE   CMTHEM       ; go back for another
                                              ;
C0EA  A9  FF      TOOHI    LDA   #255         ; make sure the minus flag is on
C0EC  60                   RTS                ; return
C0ED  A9  01      TOOLOW   LDA   #1           ; plus flag
C0EF  60                   RTS
                                              ;
C0F0  A2  03      MIDTOP   LDX   #3           ; copy from MID to TOP
C0F2  D0  02               BNE   ALWAYS       ; go forward
C0F4  A2  01      MIDBOT   LDX   #1           ; else copy from MID to BOT
C0F6  A0  01      ALWAYS   LDY   #1
C0F8  B9  E2  C1  ALWLOP   LDA   MID,Y
C0FB  9D  DE  C1           STA   BOT,X        ; .X is either 3 or 1 to start
C0FE  CA                   DEX                ; count down
C0FF  88                   DEY
C100  10  F6               BPL   ALWLOP       ; go back
C102  4C  5C  C0           JMP   MIDSET       ; set a new MID and (maybe) return
                                              ;
C105  AD  E2  C1  MIDMIN   LDA   MID          ; subtract 2 from MID
C108  38                   SEC
C109  E9  02               SBC   #2
C10B  8D  E2  C1           STA   MID
C10E  AD  E3  C1           LDA   MID+1
C111  E9  00               SBC   #0
C113  8D  E3  C1           STA   MID+1
C116  60                   RTS
                                              ;
C117  AD  E2  C1  MIDPLS   LDA   MID          ; add 2 to MID
C11A  18                   CLC
C11B  69  02               ADC   #2
C11D  8D  E2  C1           STA   MID
C120  AD  E3  C1           LDA   MID+1
C123  69  00               ADC   #0
C125  8D  E3  C1           STA   MID+1
C128  60                   RTS
                                              ;
C129              READFILE =     *
C129              SETLFS   =     65466
C129              SETNAM   =     65469
C129              OPEN     =     65472
C129              CHKIN    =     65478
```

```
C129                CHRIN    =    65487
C129                CLOSE    =    65475
C129                CLRCHN   =    65484
                                                 ;
C129  A9 01                  LDA  #1             ; logical file number
C12B  A2 08                  LDX  #8             ; device number for disk drive
C12D  A0 02                  LDY  #2             ; secondary address (2–14 are OK)
C12F  20 BA FF               JSR  SETLFS
C132  A9 08                  LDA  #FNLEN         ; length of filename
C134  A2 D2                  LDX  #<FNAME        ; address of filename
C136  A0 C1                  LDY  #>FNAME
C138  20 BD FF               JSR  SETNAM
C13B  20 C0 FF               JSR  OPEN
C13E  A2 01                  LDX  #1             ; logical file number
C140  20 C6 FF               JSR  CHKIN          ; set for input
                                                 ;
C143  A9 00                  LDA  #<BUFFER       ; set up a pointer
C145  85 FB                  STA  ZP             ; in ZP
C147  8D 00 50               STA  POINTR         ; and in the pointer table
C14A  A9 60                  LDA  #>BUFFER       ; high byte
C14C  85 FC                  STA  ZP+1
C14E  8D 01 50               STA  POINTR+1
                                                 ;
C151  A9 00                  LDA  #<POINTR       ; POINTR points to the buffer
C153  8D DA C1               STA  SB             ; put it in the starting byte SB
C156  18                     CLC
C157  69 02                  ADC  #2             ; add 2
C159  85 FD                  STA  Z2             ; and store in Z2
C15B  A9 50                  LDA  #>POINTR       ; high byte
C15D  8D DB C1               STA  SB+1           ; into SB
C160  69 00                  ADC  #0             ; handle the carry
C162  85 FE                  STA  Z2+1
C164  A0 00                  LDY  #0
C166  20 CF FF      GETCHR   JSR  CHRIN          ; get a character
C169  C9 0D                  CMP  #13            ; check for <RETURN>
C16B  F0 35                  BEQ  DELIMIT
C16D  C9 20                  CMP  #32            ; look for a space
C16F  90 09                  BCC  CHKEND         ; eliminate characters 0–31
C171  F0 2F                  BEQ  DELIMIT        ; spaces are delimiters
C173  91 FB                  STA  (ZP),Y
C175  C8                     INY
C176  D0 02                  BNE  CHKEND         ; check for the end
C178  E6 FC                  INC  ZP+1           ; increment the pointer
C17A  A6 90         CHKEND   LDX  STATUS
C17C  F0 E8                  BEQ  GETCHR         ; if equal, get more characters
C17E  A9 00                  LDA  #0             ; close it up with three zeros
C180  91 FB                  STA  (ZP),Y         ; store it
C182  20 B0 C1               JSR  ADDYZP         ; reset ZP
C185  91 FB                  STA  (ZP),Y
C187  C8                     INY
C188  91 FB                  STA  (ZP),Y
C18A  A9 01                  LDA  #1
C18C  20 C3 FF               JSR  CLOSE          ; close the file
C18F  20 CC FF               JSR  CLRCHN         ; clear channels
C192  A5 FD                  LDA  Z2             ; save the end of the buffer
C194  38                     SEC
C195  E9 06                  SBC  #6             ; which is six bytes too high
C197  8D DC C1               STA  EB             ; in end buffer EB
C19A  A5 FE                  LDA  Z2+1
C19C  E9 00                  SBC  #0
C19E  8D DD C1               STA  EB+1
C1A1  60                     RTS                 ; the end of the routine
                                                 ;
C1A2  C0 00         DELIMIT  CPY  #0             ; Is this the first character?
```

```
C1A4  F0  D4                 BEQ   CHKEND       ; yes, go back
                                                ;
                                                ; Enter this routine if a space or carriage
                                                ; return is found after a word.
C1A6  A9  00                 LDA   #0           ; zero marks the division
C1A8  91  FB                 STA   (ZP),Y       ; put a zero in memory
C1AA  20  B0  C1             JSR   ADDYZP       ; add .Y to ZP (plus one)
C1AD  4C  7A  C1             JMP   CHKEND       ; and check for end of file
                                                ;
C1B0  38            ADDYZP   SEC                ; add one to .Y
C1B1  98                     TYA                ; put it in .A
C1B2  65  FB                 ADC   ZP           ; add to ZP
C1B4  85  FB                 STA   ZP           ; fix ZP
C1B6  A9  00                 LDA   #0           ; handle the high byte
C1B8  A8                     TAY                ; put zero back into .Y
C1B9  65  FC                 ADC   ZP+1         ; add
C1BB  85  FC                 STA   ZP+1         ; and store
C1BD  C8                     INY                ; store the high byte of ZP
C1BE  91  FD                 STA   (Z2),Y       ; into the POINTR table
C1C0  88                     DEY                ; and the low byte
C1C1  A5  FB                 LDA   ZP
C1C3  91  FD                 STA   (Z2),Y
C1C5  A5  FD                 LDA   Z2           ; now add 2
C1C7  18                     CLC
C1C8  69  02                 ADC   #2
C1CA  85  FD                 STA   Z2           ; to Z2, the pointer to POINTR
C1CC  90  02                 BCC   BYEBYE       ; if carry set
C1CE  E6  FE                 INC   Z2+1         ; increment the high byte
C1D0  98            BYEBYE   TYA                ; exit with zero in .A
C1D1  60                     RTS
                                                ;
C1D2  46  49  4C    FNAME    .ASC  "FILE.S,R"   ; name of ASCII text file to be sorted and
                                                ; searched
C1DA                FNLEN    =     *-FNAME
C1DA  00  00        SB       .BYTE 0,0
C1DC  00  00        EB       .BYTE 0,0
C1DE  00  00        BOT      .BYTE 0,0
C1E0  00  00        TOP      .BYTE 0,0
C1E2  00  00        MID      .BYTE 0,0
                                                ;
C1E4                ALPHA    =     *            ; this routine alphabetizes the list of pointers
C1E4  AD  DC  C1             LDA   EB           ; set up the top of the bubble sort
C1E7  8D  74  C2             STA   ENDBUB       ; save it
C1EA  AD  DD  C1             LDA   EB+1         ; high byte
C1ED  8D  75  C2             STA   ENDBUB+1     ; too
C1F0  AD  DA  C1    BUBLP1   LDA   SB
C1F3  85  F9                 STA   P1           ; set up a zero-page pointer to the pointer
                                                ; table
C1F5  AD  DB  C1             LDA   SB+1
C1F8  85  FA                 STA   P1+1         ; P1 is the pointer to pointers
C1FA  A9  2A                 LDA   #"*
C1FC  20  D2  FF             JSR   CHROUT       ; print an asterisk
C1FF  A0  03        BUBLP2   LDY   #3           ; get two two-byte pointers (0–3)
C201  B1  F9        ZLOOPY   LDA   (P1),Y       ; get a pointer
C203  AA                     TAX                ; can't store zero page,Y from .A, but .X
                                                ; works
C204  96  FB                 STX   ZP,Y         ; not indirect
C206  88                     DEY                ; loop
C207  10  F8                 BPL   ZLOOPY       ; go back for more
                                                ; Now ZP and Z2 point to words.
C209  C8                     INY                ; .Y was 255; make it 0
C20A  B1  FB        BUBLP3   LDA   (ZP),Y       ; compare the words
C20C  D1  FD                 CMP   (Z2),Y
```

```
C20E  D0 04            BNE   CHECKM      ; if not equal, check whether they should
                                         ; swap
C210  C8               INY               ; otherwise, INC the Y register
C211  D0 F7            BNE   BUBLP3      ; and go back for more (should branch
                                         ; always)
C213  18               CLC               ; just in case
C214  90 15   CHECKM   BCC   OKRITE      ; if carry is clear, they're OK
C216  A0 00            LDY   #0          ; else, switch them
C218  A5 FD            LDA   Z2          ; put pointer in Z2
C21A  91 F9            STA   (P1),Y      ; into the pointer table
C21C  C8               INY               ; .Y is 1
C21D  A5 FE            LDA   Z2+1        ; high byte, too
C21F  91 F9            STA   (P1),Y
C221  C8               INY               ; .Y is 2
C222  A5 FB            LDA   ZP          ; and move ZP up two bytes
C224  91 F9            STA   (P1),Y
C226  C8               INY               ; .Y is 3
C227  A5 FC            LDA   ZP+1        ; high byte
C229  91 F9            STA   (P1),Y
                                         ;
                                         ; P1 has to move up a couple of notches.
C22B  A5 F9   OKRITE   LDA   P1
C22D  18               CLC
C22E  69 02            ADC   #2
C230  85 F9            STA   P1
C232  A5 FA            LDA   P1+1
C234  69 00            ADC   #0
C236  85 FA            STA   P1+1
C238  CD 75 C2         CMP   ENDBUB+1    ; are we at the end?
C23B  90 C2            BCC   BUBLP2      ; no
C23D  D0 09            BNE   ENDPASS     ; yes, move ahead
C23F  A5 F9            LDA   P1          ; maybe, check the low byte
C241  CD 74 C2         CMP   ENDBUB      ; are they the same?
C244  F0 02            BEQ   ENDPASS     ; yes, quit
C246  90 B7            BCC   BUBLP2      ; no, it's smaller
                                         ;
                                         ; End of a pass. Move ENDBUB down by
                                         ; two.
C248  AD 74 C2 ENDPASS LDA   ENDBUB
C24B  38               SEC
C24C  E9 02            SBC   #2          ; subtract 2 (low byte)
C24E  8D 74 C2         STA   ENDBUB      ; save it
C251  AD 75 C2         LDA   ENDBUB+1    ; adjust high byte
C254  E9 00            SBC   #0          ; subtract 0 (or 1)
C256  8D 75 C2         STA   ENDBUB+1
C259  CD DB C1         CMP   SB+1        ; are we down to the start?
C25C  F0 05            BEQ   MAYBE       ; maybe
C25E  90 0E            BCC   OUTBUB      ; yes, gone too far
C260  4C F0 C1         JMP   BUBLP1      ; no, jump back
C263  AD 74 C2 MAYBE   LDA   ENDBUB      ; check low
C266  CD DA C1         CMP   SB          ; against SB
C269  F0 03            BEQ   OUTBUB      ; equal, we're done
C26B  4C F0 C1         JMP   BUBLP1      ; no, keep going
C26E  A9 93   OUTBUB   LDA   #147        ; clear
C270  20 D2 FF         JSR   CHROUT      ; the screen
C273  60               RTS               ; and quit
                                         ;
C274  00 00   ENDBUB   .BYTE 0,0
C276          SEARCH   =     *           ; leave 256 bytes for this buffer
```

See also ALPNTR, ALSWAP, SRCLIN.

Name

Linear search for a string or other value

Description

Word processors often feature a *find* or a *search-and-replace* option. **SRCLIN** looks for a matching string by starting at the beginning and searching forward until the target string is discovered. A second entry point for the routine provides a *find-next-occurrence* function.

Prototype

1. Before calling the subroutine, store the start and end of text in the variables TEXSTA and TEXEND.
2. Store a search string in memory (at STRING), terminated by a zero byte.
3. Begin **SRCLIN** by setting WHERE to the start of text (TEXSTA). Skip this step if you're searching for the next occurrence.
4. Copy the pointer from WHERE to zero page (Z1).
5. Set .Y to zero.
6. Compare the character from STRING to the character pointed to by Z1 (both indexed by .Y).
7. If they're not equal, increment Z1, make sure it doesn't go past TEXEND, and loop back to step 5.
8. If Z1 exceeds TEXEND, the string hasn't been found. Store zeros into WHERE and quit.
9. If the first (or second or third) character matches, increment .Y and go back to step 6 until the zero-terminator appears.

Explanation

Compared with **SRCBIN**, this is a slow and inefficient way to look for a string in memory. But that's not necessarily a disadvantage.

In a data-oriented application such as a database program, you expect certain fields to be alphabetized. If you need a search routine, **SRCBIN** is much faster than **SRCLIN** as long as the data has already been sorted.

But in text-oriented software such as a word processor, the words in memory will be arranged grammatically instead of alphabetically. A binary search is faster than a sequential/ linear search, but you'd have to waste time and memory alphabetizing the words in the text file before the binary routine could even begin. A linear search can start searching immediately.

The **SRCLIN** routine has two entry points. If you want to search from the beginning of the text area, JSR **SRCLIN**. But if you've found the first occurrence of the string and you want to find the second, third, fourth, and so on, JSR **SRCNEX**. When the **SRCLIN** and/or **SRCNEX** routines are finished, you can find the address of the string in WHERE, in Z1, and in the A and X registers.

Warning: The **SRCLIN** routine, as it's written, is sensitive to the case of characters. For example, if you're looking for *elephant* and the word *Elephant* appears as the first word in a sentence, **SRCLIN** won't consider them a match. A capital *E* isn't the same as a lowercase *e*. To ameliorate this problem, you can insert one of the conversion routines such as **MIXUPP** to convert strings to uppercase or lowercase.

Routine

```
C000                Z1      =    $FB
C000                CHROUT  =    $FFD2
C000                LINPRT  =    $BDCD       ; LINPRT = $8E32 on the 128
                                             ;
C000  20 2B C0              JSR  SRCLIN      ; search for the string
C003  20 CD BD      BIGLOP  JSR  LINPRT      ; print the address
C006  A9 20                 LDA  #32         ; and a space
C008  20 D2 FF              JSR  CHROUT      ; after the number
C00B  AD 8F C0              LDA  WHERE       ; now check if not found
C00E  0D 8F C0              ORA  WHERE       ; if either is nonzero
C011  D0 01                 BNE  ITSOK       ; continue
C013  60                    RTS              ; else, we're finished
C014  A0 00         ITSOK   LDY  #0
C016  B1 FB         PRLOOP  LDA  (Z1),Y
C018  20 D2 FF              JSR  CHROUT
C01B  C8                    INY
C01C  C0 0A                 CPY  #10         ; print ten characters
C01E  D0 F6                 BNE  PRLOOP
C020  A9 0D                 LDA  #13         ; print RETURN
C022  20 D2 FF              JSR  CHROUT
C025  20 3E C0              JSR  SRCNEX      ; search for the next one
C028  4C 03 C0              JMP  BIGLOP
                                             ;
C02B                SRCLIN  =    *           ; beginning of the routine
C02B  AD 8B C0              LDA  TEXSTA      ; starting address of text
C02E  8D 8F C0              STA  WHERE       ; into the WHERE pointer
C031  85 FB                 STA  Z1          ; and Z1
C033  AD 8C C0              LDA  TEXSTA+1    ; high byte
C036  8D 90 C0              STA  WHERE+1     ; also
C039  85 FC                 STA  Z1+1
C03B  4C 4B C0              JMP  SRCLOP      ; skip over the next part
                                             ;
C03E                SRCNEX  =    *           ; entry for SRCNEX—search for the next
                                             ; occurrence
C03E  AD 8F C0              LDA  WHERE       ; take the WHERE pointer
C041  85 FB                 STA  Z1          ; and store in Z1
C043  AD 90 C0              LDA  WHERE+1     ; high byte, too
C046  85 FC                 STA  Z1+1
C048  20 6B C0              JSR  Z1INC       ; and count forward one to avoid repeating
C04B  A0 00         SRCLOP  LDY  #0          ; come back here for more
```

```
C04D  B9 91 C0  MOCHA     LDA   STRING,Y     ; get a character
C050  F0 0E               BEQ   FOUNDIT      ; if zero, it's the end of the string and it
                                             ; matches
C052  D1 FB               CMP   (Z1),Y       ; compare it to the text
C054  F0 06               BEQ   MORECM       ; if they're equal, continue
C056  20 6B C0            JSR   Z1INC        ; otherwise, increment the Z1 pointer
C059  4C 4B C0            JMP   SRCLOP       ; and check the next character
                                             ;
C05C  C8        MORECM    INY                ; .Y increases by one
C05D  D0 EE               BNE   MOCHA        ; and go back for the next character
C05F  60                  RTS                ; this should never happen if the string is
                                             ; fewer than 255 characters
                                             ;
C060  A6 FB     FOUNDIT   LDX   Z1           ; Z1 points to the string
C062  8E 8F C0            STX   WHERE        ; copy the address to WHERE
C065  A5 FC               LDA   Z1+1
C067  8D 90 C0            STA   WHERE+1
C06A  60                  RTS
C06B  E6 FB     Z1INC     INC   Z1           ; this just increments the Z1 pointer
C06D  D0 02               BNE   DONINC       ; do the high byte if Z1 has counted up to
                                             ; zero
C06F  E6 FC               INC   Z1+1         ; high byte
C071  A5 FB     DONINC    LDA   Z1           ; see if we're done
C073  CD 8D C0            CMP   TEXEND       ; is it the same as the end address?
C076  D0 12               BNE   OUTINC       ; no, keep going
C078  A5 FC               LDA   Z1+1         ; the low byte matches
C07A  CD 8E C0            CMP   TEXEND+1     ; compare the high
C07D  D0 0B               BNE   OUTINC       ; if not equal, keep going
C07F  68        NOTFOUND  PLA                ; trash the calling address
C080  68                  PLA                ; pull the other byte
C081  A9 00               LDA   #0
C083  8D 8F C0            STA   WHERE        ; zeros mean no match
C086  8D 90 C0            STA   WHERE+1      ; in WHERE
C089  AA                  TAX
C08A  60        OUTINC    RTS                ; return (two different ways)
                                             ;
C08B  00 CC     TEXSTA    .WORD $CC00        ; starting address of the text
C08D  FF CF     TEXEND    .WORD $CFFF        ; last character
C08F  00 CC     WHERE     .WORD $CC00        ; pointer to the middle of the file
C091  46 49 4C  STRING    .ASC  "FILE"       ; name of text file to be searched
C095  00                  .BYTE 0
```

See also SRCBIN.

Name

Store system memory to expansion RAM

Description

STASH (in conjunction with **FETCH**) provides a simple RAMdisk for the 128. On a 128, with this routine and a RAM Expansion Module (either model 1700 or 1750), you can store the contents of a block of system memory into expansion RAM.

Prototype

1. Enter this routine with the REC registers set with the appropriate system-memory base address, expansion-RAM base address, and number of bytes to transfer. The X register should contain the system bank number.
2. Load .Y with the value required in the command register (location 57089) to perform a stash operation.
3. JMP to the Kernal routine DMACALL.

Explanation

When a model 1700 or 1750 RAM Expansion Module is plugged into the 128, the RAM Expansion Controller chip (REC) in the unit appears at locations 57088–57098 in the 128's address space. This chip performs four different memory management operations. One of these—storing system memory to expansion RAM, or *stashing*—is carried out by this routine. (A discussion of the REC registers can be found in *Mapping the Commodore 128* from COMPUTE! Publications).

The program below relies on **STASH** to store the BASIC program currently in memory to one of four 32K blocks, or partitions, within the RAM expansion module. In order to insure later retrieval of the BASIC program (see the program provided with **FETCH**), certain pointers—specifically to the start and end of the program—are saved before the program itself.

To use the program listed here, assemble it and SYS to its starting address from BASIC. Following the SYS address, specify the partition where the current BASIC program is to be saved. For instance, assuming you assemble the program at 3072 as shown, you would enter **SYS3072,1** to store a BASIC program in partition 1.

When the SYS executes, BASIC stores the partition number you've specified in the accumulator. At this point, the machine language program takes over.

First, it checks to see that the partition number provided is in the range 1–4. If it isn't, an error message to this effect is printed and the program terminates. Otherwise, the program continues by setting up the REC registers. The first one considered is the expansion bank register.

The two memory expansion modules currently available are partitioned into 64K blocks, or banks, of free RAM. The model 1700 has two banks (banks 0 and 1), for a total of 128K while the 1750 has eight banks (banks 0–7), for a total of 512K. Since the program here requires four separate 32K blocks of memory, banks 0 and 1 are used in the RAM expansion module, with partitions 1 and 2 assigned to bank 0, and partitions 3 and 4 to bank 1.

After the proper expansion bank number has been stored, the base address for the expansion RAM module is set to either 0K or 32K. Following this, the system base address (ZP) to the BASIC pointers, number of bytes to stash (4), and the sytem bank number (0) are stored in the appropriate REC registers. **STASH** is then called.

STASH, in turn, accesses DMACALL, a Kernal routine that is generally called when performing operations involving expansion RAM. The requested REC command—the value ordinarily placed in 57089—is passed to DMACALL in the Y register.

Once the start- and end-of-BASIC pointers have been stashed, the BASIC program itself is stored in the same partition with a similar procedure. During the stash operation, the expansion-RAM base address increments automatically as each byte is transferred (bits 6 and 7 in 57098 are 00 by default). As a result, once the BASIC pointers have been stored, the expansion base address is ready for a second stash operation and requires no updating.

Note: A swap or verify routine would closely resemble the setup shown in this program. If you attempt to write one, be sure to change the contents of the command register (in .Y) for the proper operation (stash, fetch, swap, or verify) before calling DMACALL.

Routine

```
0C00        CHROUT   =   65490
0C00        DMACALL  =   65360     ; Kernal routine which passes command in .X
                                   ; to DMA controller
0C00        DMASYA   =   57090     ; DMA system memory base address register
0C00        DMAEXA   =   57092     ; DMA expansion memory base address
                                   ; register
```

```
0C00                  DMABNK   =   57094        ; DMA expansion memory bank register
0C00                  DMADAT   =   57095        , DMA number of bytes to transfer
0C00                  TXTTAB   =   45           ; start-of-BASIC pointer
0C00                  TEXTTP   =   4624         ; end-of-BASIC program pointer
0C00                  ZP       =   251
                                                ;
                                                ; Store BASIC program into RAM expansion
                                                ; bank 0 or 1 on 32K boundaries.
                                                ; Use this program along with the program
                                                ; under FETCH entry.
0C00  C9 01                    CMP #1           ; make sure .A is in range 1-4
0C02  90 5D                    BCC PRTMSG       ; .A is less than 1, so print an error message
                                                ; and leave
0C04  C9 05                    CMP #5
0C06  B0 59                    BCS PRTMSG       ; .A is 5 or greater, so print error message
                                                ; and leave
0C08  38                       SEC              ; now subtract 1 to put it in range 0-3
0C09  E9 01                    SBC #1
0C0B  4A                       LSR              ; determine RAM expansion bank
0C0C  8D 06 DF                 STA DMABNK       ; store it into DMA bank register
0C0F  A9 00                    LDA #0           ; determine 32K offset in each bank (high
                                                ; byte)
0C11  8D 04 DF                 STA DMAEXA       ; also store zero into base address for
                                                ; expansion memory (low byte)
0C14  90 02                    BCC EXPOFF       ; if partition number is 1 or 3, carry is clear,
                                                ; so 0K offset
0C16  A9 20                    LDA #32          ; offset by 32K if partition number is 2 or 4
0C18  8D 05 DF EXPOFF          STA DMAEXA+1     ; store in base address for expansion memory
                                                , (high byte)
0C1B  A5 2D                    LDA TXTTAB       , save start-of-BASIC address pointer in zero
                                                ; page
0C1D  85 FB                    STA ZP
0C1F  A5 2E                    LDA TXTTAB+1
0C21  85 FC                    STA ZP+1
0C23  AD 10 12                 LDA TEXTTP       ; save end-of-BASIC address pointer in zero
                                                ; page
0C26  85 FD                    STA ZP+2
0C28  AD 11 12                 LDA TEXTTP+1
0C2B  85 FE                    STA ZP+3
0C2D  A9 FB                    LDA #ZP          ; store starting address of two pointers in
                                                ; system memory address register
0C2F  8D 02 DF                 STA DMASYA       ; low byte
0C32  A9 04                    LDA #4           ; store number of bytes to transfer in DMA
                                                , register (low byte)
0C34  8D 07 DF                 STA DMADAT
0C37  A9 00                    LDA #0           ; store zero to high byte
0C39  8D 08 DF                 STA DMADAT+1
0C3C  8D 03 DF                 STA DMASYA+1     ; also store zero to high byte of system
                                                ; memory address
0C3F  AA                       TAX              ; put system memory bank number in .X
0C40  20 6F 0C                 JSR STASH        ; store BASIC pointers
                                                ; Now store BASIC program directly after the
                                                , pointers.
0C43  38                       SEC              , determine number of bytes in BASIC
                                                ; program
0C44  AD 10 12                 LDA TEXTTP       ; get end of BASIC low byte
0C47  E5 2D                    SBC TXTTAB       ; subtract start-of-BASIC low byte
0C49  8D 07 DF                 STA DMADAT       , store result into DMA register for number of
                                                ; bytes to transfer
0C4C  AD 11 12                 LDA TEXTTP+1     ; get end-of-BASIC high byte
0C4F  E5 2E                    SBC TXTTAB+1     ; subtract start-of-BASIC high byte
0C51  8D 08 DF                 STA DMADAT+1     ; store to high byte of register
0C54  A5 2D                    LDA TXTTAB       ; store starting address of BASIC as system
                                                ; base address
```

```
0C56  8D 02 DF              STA   DMASYA
0C59  A5 2E                 LDA   TXTTAB+1
0C5B  8D 03 DF              STA   DMASYA+1
                                               ; System bank number is in .X, DMAEXA
                                               ; updates automatically (see 57098).
0C5E  4C 6F 0C              JMP   STASH        ; store BASIC program and RTS
                                               ;
0C61  A0 00       PRTMSG    LDY   #0           ; index for PRTLOP
0C63  B9 74 0C    PRTLOP    LDA   ERRMSG,Y     ; get a character for the error message
0C66  F0 06                 BEQ   PRTEND       ; end on a zero byte
0C68  20 D2 FF              JSR   CHROUT       ; print the character if not zero
0C6B  C8                    INY                ; next character
0C6C  D0 F5                 BNE   PRTLOP       ; branch always
0C6E  60          PRTEND    RTS                ; leave the program
                                               ;
                                               ; Enter this routine with DMA registers set
                                               ; up, and system bank number in .X
0C6F  A0 80       STASH     LDY   #%10000000   ; command register (57089) value for stash
0C71  4C 50 FF              JMP   DMACALL      ; call DMA Kernal routine and RTS
                                               ;
0C74  4E 4F 54    ERRMSG    .ASC  "NOT A VALID PARTITION NUMBER"
                                               ; error message
0C90  00                    .BYTE 0            ; terminator byte
```

See also FETCH.

Name

Print a string on the 64 with STROUT

Description

STP64 relies on the BASIC routine STROUT to print a string to the current output device.

Prototype

1. Load the address of the string into .A (low byte) and .Y (high byte).
2. JSR to the STROUT routine in BASIC ROM to print the string (ending in a zero byte).

Explanation

Due to the limits of STROUT, **STP64** can print strings that are no longer than 255 bytes. Use **STRCPT** if you wish to print longer strings.

In the example, **STP64** sends the string to the screen (the default device). Output can be directed to other peripherals, such as printers, by changing the current output device number (location 154) or by calling the Kernal CHKOUT routine after opening a file to another device.

Warning: Be sure to place the string you intend to print outside your working code. If you place the string immediately after the JSR STROUT instruction, the 64 will interpret the characters of the string as if they were ML instructions.

Routine

```
C000              STROUT   =      43806
                                                ;
                                                ; Print string "HELLO".
C000  A9 08       STP64    LDA    #<STRING     ; low byte of string
C002  A0 C0                LDY    #>STRING     ; high byte of string
C004  20 1E AB             JSR    STROUT       ; print the string
C007  60                   RTS
C008  48 45 4C    STRING   .ASC   "HELLO"      ; message to print
C00D  00                   .BYTE  0            ; ending in a zero byte
```

See also PTABAD, PTABCT, STP128, STRCPT, STRLEN.

Name

Print a string on the 128 with PRIMM

Description

STP128 relies on the Kernal routine PRIMM to print a string to the current output device.

Prototype

1. JSR to PRIMM.
2. The ASCII string (ending in a zero byte) immediately follows in the code.

Explanation

Because it relies on PRIMM, STP128 can only print strings that are no longer than 255 bytes. To print longer strings, use **STRCPT**.

In the example, **STP128** sends output to the screen (the default device). Output can be directed to other peripherals, such as printers, by changing the current output device number in location 154 or by opening a channel and performing a Kernal CHKOUT.

Warning: Always JSR to PRIMM rather than JMPing to it, since PRIMM uses the return address of the JSR to locate the string.

Routine

```
0C00                PRIMM    =      65405

                                           ;
                                           ; Print HELLO.
0C00  20  7D  FF  STP128    JSR    PRIMM    ; print the string that follows
0C03  48  45  4C  STRING    .ASC   "HELLO"  ; ASCII message to print
0C08  00                    .BYTE  0        ; and ends in a zero byte
0C09  60                    RTS
```

See also PTABAD, PTABCT, STP64, STRCPT, STRLEN.

Name

Check for STOP key by using the system STOP flag

Description

The flag at location 145 is used to detect when the STOP key has been pressed. A value of 127 in this location indicates that STOP has been pressed.

Prototype

1. Compare the contents of the STOP flag with 127.
2. Return with the status register Z flag set if STOP is pressed.

Explanation

Similar to the example routine for **STPKER**, this routine prints *B*'s until STOP is pressed. Comparing the contents of STKEY with 127 sets or clears the Z flag just as if we had executed the Kernal STOP routine. That is, only if STOP is detected will Z = 1.

Note: The flag at 145 is updated only during normal IRQ interrupts. So if you write your own interrupt routine, use **STPKER** instead. One advantage of using **STPFLG**, however, is that only .A is affected, whereas **STPKER** affects both .A and .X.

Routine

```
C000                STKEY   =     145        ; STOP key flag
C000                CHROUT  =     65490
                                             ;
                                             ; Print B's until stop is pressed.
C000  A9 42         LOOP    LDA   #66        ; print B
C002  20 D2 FF              JSR   CHROUT
C005  20 0B C0              JSR   STPFLG     ; check STOP key
C008  D0 F6                 BNE   LOOP       ; STOP key not pressed, so LOOP
C00A  60                    RTS
                                             ;
                                             ; Check STOP key flag. If pressed, set Z flag
                                             ; in status register.
C00B  A5 91         STPFLG  LDA   STKEY      ; check STOP key flag
C00D  C9 7F                 CMP   #127       ; STOP key pressed?
C00F  60                    RTS              ; Z flag set accordingly
```

See also SHFCHK, STPKER.

Name

Check for STOP key using Kernal STOP routine

Description

The Kernal STOP routine allows you to determine when the STOP key has been pressed. The zero flag is set if the STOP key, either alone or in combination with certain other keys, has been pressed. Otherwise, the Z flag is clear.

Prototype

1. JSR to the Kernal STOP routine and RTS (or simply JMP to STOP).
2. Upon return, the Z flag will be set if STOP is pressed.

Explanation :

To demonstrate this routine, we print *A*'s while Z = 0. When STOP is pressed, Z = 1, and we clear the screen.

Note: Unlike **STPFLG**, **STPKER** is not IRQ-dependent. However, **STPKER** affects both .A and .X, whereas **STPFLG** only affects the accumulator.

Routine

```
C000                STOP     =     65505      ; Kernal STOP routine
C000                CHROUT   =     65490
                                              ;
                                              ; Print A's. When STOP key pressed, clear
                                              ; screen.
C000  A9 41         LOOP     LDA   #65        ; print A
C002  20 D2 FF               JSR   CHROUT
C005  20 10 C0               JSR   STPKER     ; check STOP key
C008  D0 F6                  BNE   LOOP       ; if zero is clear, then LOOP
C00A  A9 93         CLRCHR   LDA   #147       ; clear screen
C00C  20 D2 FF               JSR   CHROUT
C00F  60                     RTS
                                              ;
                                              ; Check STOP key. Z flag set if pressed.
C010  20 E1 FF      STPKER   JSR   STOP       ; Kernal STOP key check
C013  60                     RTS
```

See also SHFCHK, STPFLG.

Name

Print a string with a custom printing routine

Description

This routine prints a zero-terminated ASCII string of any length. It's similar to the STROUT routine in Commodore 64 ROM.

Prototype

1. Load .A with the low byte of the address of the string and store it in zero page.
2. Do the same with the high byte of the address of the string.
3. Set an index (.Y) to zero to initialize the main loop (STRLOP).
4. Execute STRLOP until the zero byte is reached or until .Y reaches zero.
5. If the index rolls over, increment the high byte value in the zero-page pointer to the string address and continue STRLOP.

Explanation

You may find the built-in routines for printing strings (BASIC STROUT on the 64 and Kernal PRIMM on the 128) limiting in certain situations. Suppose, for instance, that while programming on your 64, you need to switch out BASIC ROM. It may not be convenient to switch BASIC back in during your program just to print a string with STROUT. Instead, you can simply incorporate **STRCPT** into your code.

Furthermore, there will be times when you'll need to print strings longer than 255 characters. Neither STROUT nor PRIMM can handle this chore. But **STRCPT**, designed to print longer strings, would be ideal.

Also, **STRCPT** is not specific to the 64 or the 128. For this reason you'll see **STRCPT** in many programs in this book.

Much like **STP64** and **STP128**, the important point to remember in using **STRCPT** is to place the string outside your working code. If you place the string in the working portion of **STRCPT**, your computer will attempt to execute the characters of the string as if they were ML instructions.

In the example, **STRCPT** sends the string to the screen (the default device). Output can be directed to other peripherals, such as printers, by opening a channel to the device and executing CHKOUT.

Routine

```
C000                CHROUT  =      65490
C000                ZP      =      251
                                          ;
                                          ; Print HELLO with custom print routine
                                          ; (allows >255 characters)
C000  A9 1A         STRCPT  LDA   #<STRING  ; low byte of string
C002  85 FB                 STA   ZP        ; store it
C004  A0 C0                 LDY   #>STRING  ; high byte of string
C006  84 FC                 STY   ZP+1      ; store it also
C008  A0 00                 LDY   #0        ; initialize index
C00A  B1 FB         STRLOP  LDA   (ZP),Y    ; load each character from string
C00C  F0 0B                 BEQ   FINISH    ; if zero byte, then finished
C00E  20 D2 FF              JSR   CHROUT    ; print character
C011  C8                    INY             ; for next character
C012  D0 F6                 BNE   STRLOP    ; if not more than 256 bytes, then get next
                                            ; character
C014  E6 FC                 INC   ZP+1      ; otherwise, increment high byte of the
                                            ; pointer
C016  4C 0A C0              JMP   STRLOP    ; and continue printing
C019  60            FINISH  RTS
                                          ;
C01A  48 45 4C      STRING  .ASC  "HELLO"   ; message to print
C01F  00                    .BYTE 0         ; ending in zero byte
```

See also PTABAD, PTABCT, STP128, STP64, STRLEN.

Name

Determine the length of a string

Description

From time to time, you'll want to find out how many characters are in a particular string. Perhaps a string-handling operation or a screen-positioning routine requires this information. **STRLEN** provides you with the length of any zero-terminated string containing fewer than 256 characters.

Prototype

1. Initialize .Y to 255 to serve as a character counter .
2. Begin counting characters in the string by incrementing .Y.
3. Check each character in the string for a zero byte.
4. If the character byte is not zero, go to step 2.
5. Otherwise, transfer the length of the string (in the Y register) to .A and RTS.

Explanation

In the example below, a line of text is entered into the text input buffer by using the BASIC routine INLIN. The address of this string data is stored in zero page. **STRLEN** then returns the length of the string in the accumulator. The framing routine prints the length with **NUMOUT** prior to returning to BASIC.

Note: An RTS cannot be used to return to BASIC here because the text in the input buffer would be interpreted by BASIC as a direct command. See **TXTINP** for a discussion of this problem.

Warning: The loop that searches for a string ($C01B–$C01F) will never end if there are no zero bytes within the 256 locations after the starting address of the buffer. The INLIN ROM routine always ends a string with the number 0, so this is not a concern within this example program. However, if you use this subroutine within your own programs, be sure the string you're examining is fewer than 256 characters long and that it ends with a zero byte.

Routine

```
C000          CHROUT  =   65490
C000          BUF     =   512
C000          ZP      =   251
C000          INLIN   =   42336     ; INLIN = 22176 on the 128
C000          LINPRT  =   48589     ; LINPRT = 36402 on the 128
                                    ;
                                    ; Input a line of text until RETURN and
                                    ; determine its length.
```

```
C000  20 60 A5            JSR   INLIN     ; input a line of text with the BASIC routine
                                          ; INLIN
                                          ;
                                          ; Store the resulting text string address in
                                          ; zero page
C003  A9 00               LDA   #<BUF     ; low byte of input buffer
C005  85 FB               STA   ZP        ; store in zero page
C007  A0 02               LDY   #>BUF     ; high byte of input buffer
C009  84 FC               STY   ZP+1      ; also store in zero page
                                          ;
C00B  20 19 C0            JSR   STRLEN    ; get string length
                                          ;
                                          ; Print length with NUMOUT.
C00E  AA         NUMOUT   TAX             ; place low byte of number in .X
C00F  A9 00               LDA   #0        ; high byte is zero
C011  20 CD BD            JSR   LINPRT    ; print the length
C014  A2 80               LDX   #128      ; error handler code for READY message
C016  6C 00 03            JMP   (768)     ; return to BASIC and print READY prompt
                                          ;
                                          ; Return the length of the string (<256
                                          ; characters) in .A.
                                          ; String's address is in zero page.
C019  A0 FF      STRLEN   LDY   #255
C01B  C8         LENLOP   INY             ; index into string
C01C  B1 FB               LDA   (ZP),Y    ; load the next character
C01E  D0 FB               BNE   LENLOP    ; check for zero byte
C020  98                  TYA             ; you've reached the end of the string, so
                                          ; return length in .A
C021  60                  RTS
```

See also PTABAD, PTABCT, STP128, STP64, STRCPT.

Name

Subtract one byte value from another

Description

The SBC (SuBtract with Carry) instruction subtracts a value from the number currently in the accumulator. The example program illustrates the basic technique for subtracting one number from another.

Prototype

1. Set the carry flag with SEC.
2. Load the accumulator (LDA) with the first number.
3. Subtract the second number (SBC) and handle the result as you wish.

Explanation

The example program waits for the user to press two keys. If C (ASCII 67) is pressed first, followed by A (ASCII 65), the number 65 is subtracted from 67 and the result (2) prints to the screen.

If you switch the two letters, the calculation of 65 − 67, (which should be −2) gives a result of 254 instead. It's important to remember that byte values are limited to the range 0–255 and that if you add or subtract two numbers that result in a number outside of that range, the values wrap around at 256. When such an overflow occurs, the carry flag will be set (after addition) or clear (after subtraction).

An interesting side effect of this fact is that the compare instructions—CMP, CPX, and CPY—which compare two numbers, act like SBC. If you subtract a smaller (or equal) number, carry is set. If you subtract a larger number, carry is clear. Thus, after a compare instruction, carry is clear if the number in .A, .X, or .Y is smaller than the second number.

Routine

```
C000                GETIN   =     $FFE4
C000                LINPRT  =     $BDCD       ; LINPRT = $8E32 on the 128
C000                CHROUT  =     $FFD2
                                              ;
C000  20  37  C0            JSR   GETKEY      ; get a key (ASCII value)
C003  8D  3D  C0            STA   NUMBER1     ; store it
C006  20  37  C0            JSR   GETKEY      ; get a second key
C009  8D  3E  C0            STA   NUMBER2     ; store it, too
C00C  AE  3D  C0            LDX   NUMBER1     ; now print it
C00F  A9  00                LDA   #0
C011  20  CD  BD            JSR   LINPRT
C014  A9  0D                LDA   #13
C016  20  D2  FF            JSR   CHROUT      ; print RETURN
C019  AE  3E  C0            LDX   NUMBER2     ; second number
```

```
C01C  A9 00              LDA    #0
C01E  20 CD BD           JSR    LINPRT      ; print it
C021  A9 0D              LDA    #13
C023  20 D2 FF           JSR    CHROUT      ; RETURN again
                                            ;
C026  AD 3D C0 SUBBYT    LDA    NUMBER1     ; the first number
C029  38                 SEC                ; set the carry flag
C02A  ED 3E C0           SBC    NUMBER2     ; subtract the second
C02D  8D 3F C0           STA    TOTAL       ; store it
C030  AA                 TAX                ; put it in .X
C031  A9 00              LDA    #0
C033  20 CD BD           JSR    LINPRT      ; and print it
C036  60                 RTS
                                            ;
C037  20 E4 FF GETKEY    JSR    GETIN
C03A  F0 FB              BEQ    GETKEY
C03C  60                 RTS
                                            ;
C03D  00       NUMBER1   .BYTE  0
C03E  00       NUMBER2   .BYTE  0
C03F  00       TOTAL     .BYTE  0
```

See also SUBFP, SUBINT.

Name

Subtract one floating-point number from another

Description

Given a number in the second floating-point accumulator (FAC2) and another number in FAC1, this routine subtracts (FAC2 minus FAC1) and puts the result in FAC1.

Prototype

1. Store a number in FAC2.
2. Store another number in FAC1.
3. Call the ROM routine FSUBT.

Explanation

The example routine subtracts 300 from 258. The result is −42, which is converted to ASCII numbers and is printed to the screen. Note the abundance of ROM routine calls, which generally make it easy to handle floating-point values.

Routine

```
C000                ZP        =    $FB
C000                CHROUT    =    $FFD2
C000                FSUBT     =    $B853     ; FSUBT = $8831 on the 128—subtract FAC1
                                             ; from FAC2; result in FAC1
C000                MOVEF     =    $BC0F     ; MOVEF = $8C3B on the 128—moves FAC1
                                             ; to FAC2
C000                GIVAYF    =    $B391     ; GIVAYF = $AF03 on the 128—converts
                                             ; integer to floating point
C000                FOUT      =    $BDDD     ; FOUT = $8E42 on the 128—converts FAC1
                                             ; to ASCII string
                                             ;
                                             ; Convert the numbers 258 and 300 to
                                             ; floating point and subtract.
C000  A9 01                   LDA  #>258     ; high byte of 258
C002  A0 02                   LDY  #<258     ; low byte
C004  20 91 B3                JSR  GIVAYF    ; convert it; now it's in FAC1
C007  20 0F BC                JSR  MOVEF     ; move FAC1 to FAC2
C00A  A9 01                   LDA  #>300     ; high byte of 300
C00C  A0 2C                   LDY  #<300     ; low byte
C00E  20 91 B3                JSR  GIVAYF    ; convert it
                                             ; FAC1 now holds 300, and FAC2 holds 258
C011  20 29 C0                JSR  SUBFP     ; subtract (258 − 300), the result (−42) is left
                                             ; in FAC1
C014  20 DD BD                JSR  FOUT      ; convert to ASCII
C017  85 FB                   STA  ZP        ; pointer
C019  84 FC                   STY  ZP+1      ; to the string
```

```
C01B  A0  00                 LDY   #0
C01D  B1  FB        PRTLOP   LDA   (ZP),Y
C01F  D0  01                 BNE   PRNIT
C021  60                     RTS
C022  20  D2  FF    PRNIT    JSR   CHROUT
C025  C8                     INY
C026  D0  F5                 BNE   PRTLOP
C028  60                     RTS
                                             ;
C029  20  53  B8    SUBFP    JSR   FSUBT     ; subtract FAC1 from FAC2
C02C  60                     RTS             ; the result is in FAC1
```

See also SUBBYT, SUBINT.

Name

Subtract one 2-byte integer from another

Description

A single opcode (SBC) handles subtraction, but you have to set the carry flag first. This routine illustrates how to do multiple-byte subtraction.

Prototype

1. Set the carry flag (SEC).
2. Load the low byte into .A (LDA).
3. Subtract with carry (SBC) the second byte.
4. Store the result (STA).
5. Repeat the LDA, SBC, STA sequence for higher bytes.

Explanation

The rule to remember for both adding and subtracting is always to clear the carry flag before adding and always to set carry before subtracting. Start with the low byte and work toward the higher bytes. The SEC (SEt Carry) instruction is needed only once at the beginning of the multiple-byte subtraction. After the first byte is subtracted, carry takes care of itself.

The example program takes the value in the pointer from VARTAB (the end of the BASIC text area) and subtracts the address of the beginning of the BASIC text area. It then prints a number that represents the number of bytes used by the BASIC program in memory. Since BASIC puts two zeros at the end of a program, the number 2 will print if you have no program in memory.

Note: If the number subtracted is larger than the other number (500 1120, for example), the carry flag will be clear when the routine finishes, and the result will wrap around from $0000 to $FFFF or below.

Routine

```
C000                TXTTAB   =     43           ; TXTTAB = 45 on the 128—beginning of
                                                ; BASIC program text
C000                VARTAB   =     45           ; end of the text for BASIC (substitute
                                                ; TXTTP = 4624 for the 128)
C000                LINPRT   =     $BDCD        ; LINPRT = $8E32 on the 128
                                                ;
C000  A5  2D                 LDA   VARTAB       ; the end of BASIC (substitute TXTTP for the
                                                ; 128)
C002  8D  35  C0             STA   NUM1
C005  A5  2E                 LDA   VARTAB + 1   ; high byte (substitute TXTTP + 1 for the 128)
C007  8D  36  C0             STA   NUM1 + 1
C00A  A5  2B                 LDA   TXTTAB       ; the start of BASIC
```

```
C00C  8D 37 C0            STA   NUM2
C00F  A5 2C               LDA   TXTTAB+1     ; high byte
C011  8D 38 C0            STA   NUM2+1
                                             ;
                                             ; The two numbers have been prepared
C014  20 21 C0            JSR   SUBINT       ; subtract the second number from the first
                                             ;
C017  AE 39 C0            LDX   MINUS        ; low byte of the result
C01A  AD 3A C0            LDA   MINUS+1      ; high byte
C01D  20 CD BD            JSR   LINPRT       ; print it
C020  60                  RTS
                                             ;
C021  38         SUBINT   SEC                ; always set carry before subtracting
C022  AD 35 C0            LDA   NUM1         ; low byte first
C025  ED 37 C0            SBC   NUM2         ; subtract
C028  8D 39 C0            STA   MINUS        ; and store the result
C02B  AD 36 C0            LDA   NUM1+1       ; high byte
C02E  ED 38 C0            SBC   NUM2+1       ; subtract (don't SEC)
C031  8D 3A C0            STA   MINUS+1
C034  60                  RTS                ; finished
                                             ;
C035  00 00      NUM1     .BYTE 0,0
C037  00 00      NUM2     .BYTE 0,0
C039  00 00      MINUS    .BYTE 0,0
```

See also SUBBYT, SUBFP.

Name

Save processor registers in memory

Description

At times you'll face a situation where you'll need to go to a subroutine that might change the contents of the processor registers .A, .X, .Y, and .P—but you want to remember the current state of the registers when the subroutine ends. This routine saves the registers in memory, so you can find them again when you return.

Prototype :

1. Push .P onto the stack temporarily.
2. Store .A, .X, and .Y in memory.
3. Pull .P from the stack, but into .A (PLA, not PLP).
4. Store .A into memory.

Explanation

The processor status register contains the various flags—zero, negative, overflow, carry, and so on—and the flags can change very quickly. (A single LDA will often change several flags.) Because it's so fragile, it must be handled first. After we have pushed it temporarily onto the stack, the rest of the subroutine is fairly simple. Just store the registers into memory: TEMPA, TEMPX, and TEMPY. Finally, the P register is pulled off the stack (into the accumulator this time), and it's stashed in TEMPP.

Note: This routine is slower and takes more memory than the routine that saves the registers onto the stack. It does have one advantage, though: This one can exist as a subroutine. You can JSR **SVREGM** before calling the routine that changes the registers. The other routine must be in-line code. If you have several areas where the registers must be remembered, this subroutine will save memory in the long run. On the other hand, if you find yourself constantly saving and restoring the registers, your program design may be flawed; this sort of routine can be replaced by various other techniques.

Routine

```
C000  08          SVREGM  PHP          ; first push the .P status to retrieve later
C001  8D  0F  C0          STA   TEMPA  ; save .A
C004  8E  10  C0          STX   TEMPX  ; save .X
C007  8C  11  C0          STY   TEMPY  ; save .Y
C00A  68                  PLA          ; get .P from the stack (into .A this time)
C00B  8D  12  C0          STA   TEMPP  ;
```

```
C00E  60                  RTS          ; we're done
                                       ;
                                       ; Variables
C00F  00          TEMPA   .BYTE  00
C010  00          TEMPX   .BYTE  00
C011  00          TEMPY   .BYTE  00
C012  00          TEMPP   .BYTE  00
```

See also RSREGM, SVREGS.

Name

Save and restore registers on the stack within a routine (in-line code)

Description

Occasionally, you'll have a situation where the A, X, and Y registers hold important information, but you'll need to call a subroutine that may leave them in an indeterminate state. The solution is to save them as you enter the routine and then restore them before exiting. The fastest way to store registers is to push them onto the stack.

Prototype

1. Push .P (processor status) onto the stack.
2. Push .A and then transfer .X and .Y to .A for pushing.
3. Execute the routine.
4. Restore the registers by pulling them off the stack (in reverse order).

Explanation

The processor status contains the various flags (.N, .Z, .C, and so forth) and can change with a single LDA, so we have to push it first (PHP). Next, we have to save the accumulator, because it's not possible to push .X and .Y directly. After .P and .A have been saved, .X is transferred to .A (TXA) and pushed (PHA), and then .Y is transferred and pushed.

The four important registers are now on the stack. The routine at $C006–$C01E is unimportant (it prints the letters *A–Z*), but it does mess up the contents of all registers. So, when it's finished, we get back the registers by pulling the values back. Since they went on the stack in the order .P, .A, .X, and .Y, it's necessary to pull them off in the reverse order (.Y, .X, .A, and .P). When that's done, the RTS sends us back to the calling routine.

Warning: You must do the pushing and pulling within the same routine. The **SVREGS** routine cannot be used as a separate subroutine because JSR needs the stack to preserve the program counter. If you were to use **SVREGS** as a subroutine, the JSR would put two bytes onto the stack; then **SVREGS** would push .P, .A, .X, and .Y onto the stack. The RTS would cause two bytes to be pulled off (the return address), but they would be the former contents of .X and .Y, and the program would return to some unknown location.

In general, if you push a certain number of bytes onto the stack within a subroutine, you must pull the same number off before you RTS.

Routine

```
C000                CHROUT   =    $FFD2
                                              ;
C000  08            SVREGS   PHP              ; push the processor status, which is most
                                              ; fragile
C001  48                     PHA              ; push the accumulator, because we need it
                                              ; for the next two pushes
C002  8A                     TXA              ; .X into .A
C003  48                     PHA              ; push it
C004  98                     TYA              ; .Y into .A
C005  48                     PHA              ; push it
                                              ; .P, .A, .X, and .Y have been pushed onto
                                              ; the stack
                                              ; in that order.
                                              ; now a dummy routine, just to change the
                                              ; registers
C006  A2 02                  LDX   #2         ; .X is changed
C008  A9 41                  LDA   #65        ; .A is changed
C00A  A0 0D         OUTLOP   LDY   #13        ; .Y is changed
C00C  20 D2 FF      INLOOP   JSR   CHROUT     ; print it
C00F  18                     CLC              ; .P is changed
C010  69 01                  ADC   #1         ; increase the accumulator
C012  88                     DEY              ; count down 13 to 1
C013  D0 F7                  BNE   INLOOP     ; print 13 characters
C015  48                     PHA              ; save .A (a save within a save)
C016  A9 0D                  LDA   #13        ; carriage return
C018  20 D2 FF               JSR   CHROUT     ; new line
C01B  68                     PLA              ; get back .A
C01C  CA                     DEX
C01D  D0 EB                  BNE   OUTLOP     ; go back for the second 13 letters
                                              ;
                                              ; By now the registers have been
                                              ; changed, so we restore them in
                                              ; reverse order (.Y, .X, .A, .P).
C01F  68                     PLA              ; pull
C020  A8                     TAY              ; put it in .Y
C021  68                     PLA              ; pull
C022  AA                     TAX              ; into .X
C023  68                     PLA              ; pull .A
C024  28                     PLP              ; pull .P
C025  60                     RTS              ; return, with all registers intact
```

See also RSREGM, SVREGM.

Name

Memory swap

Description

Whenever you need to swap two blocks of memory, use this routine. On the 128, **SWAPIT** can even exchange memory from one bank to another.

Prototype

This is a two-part routine. In an initialization routine (here, either SWAPCO or SWAPSC):

1. Store the starting address of the lower memory block to be swapped in ZP and the address of the higher memory block in ZP+2.
2. The subroutine ONELES, called from SWAPCO and SWAPSC, insures that the memory block pointed to by ZP has the lower address of the two blocks to be swapped. (If the address of the memory block in ZP is higher, a second subroutine called FLIPZP switches the addresses in ZP and ZP+2.)

In **SWAPIT** itself:

1. Jump to the subroutine OVRLAP to determine whether the two memory blocks overlap. In the process, store the number of bytes to be swapped in a counter (COUNTR).
2. If the two memory blocks overlap, return from OVRLAP with the carry flag set to indicate that an error has occurred.
3. Continue with **SWAPIT** if the carry is clear (meaning there is no overlap). Otherwise, return to the main calling program with the carry set.
4. Load a byte from the first block. Store it in .X temporarily while a byte is read from the second memory block.
5. Store the byte from the second block into the first. Recall the byte in .X and store it into the second memory block.
6. Repeat steps 4 and 5 until the bytes counter (COUNTR) reaches zero.
7. Clear the carry flag before returning from **SWAPIT**.

Explanation

In the example program, blocks of memory representing the screen are exchanged—first color and then text memory. You could use a routine like this one in setting up a help screen. Whenever the user pressed a certain key, the help screen

would be swapped with the current screen. Later, the normal screen would be reenabled.

Enter any key within the main loop (MAINLP) of this program, and the corresponding character prints to the screen. The exceptions are the F1, F7, and left-arrow (←) key. Left arrow exits the program, while F1 and F7 cause screen swaps. F1 saves the current screen as a help screen (as long as HELPFL = 0) and F7 retrieves it. Once the help screen is displayed, any key you press restores the normal text screen.

On the 128, since the function keys are predefined as BASIC commands, you'll need to enter the following line before running the program:

```
KEY1,CHR$(133): KEY7,CHR$(136)
```

A number of subroutines are called in preparation for **SWAPIT**. The first one (either SWAPCO or SWAPSC, depending on whether you're swapping color or text memory) stores the addresses of the two memory blocks to swap in zero page. Before exiting this routine, a second subroutine, ONELES, is accessed. ONELES (calling the subroutine FLIPZP if it's needed) insures that the address pointed to by the first zero-page pointer (ZP) is lower in memory than that in the second zero-page pointer (ZP+2).

Once the pointers are created, **SWAPIT** is called. The first thing **SWAPIT** does is check for overlap between the two blocks of memory that are going to be swapped. This is handled by the subroutine OVRLAP.

OVRLAP initially stores the number of bytes you want to swap—previously defined as NUMBER—in a two-byte counter (COUNTR). At the same time, it adds this number to the block that's lower in memory (in ZP). If the resulting number is higher than the start of the second memory block, the carry flag is set to indicate overlap. So, upon returning to **SWAPIT**, if carry is set, an error message is printed, and the program terminates.

If there's no overlap, **SWAPIT** continues, exchanging bytes one at a time from the two memory blocks until COUNTR decrements to zero.

On the 128, memory can be swapped from bank to bank. Two Kernal routines specific to the 128 are required: INDFET, in place of the LDA (ZP),Y at $C095, and INDSTA, for the

STA (ZP),Y at $C09A. In each case, you must substitute either three or four instructions. Look at **MVU128** or **MOVEDN** for details on how to set this up.

Routine

```
C000                ZP       =     251
C000                CHROUT   =     65490
C000                GETIN    =     65508
C000                BLOCK1   =     1024      ; memory block 1
C000                COLBL1   =     55296     ; color block 1
C000                BLOCK2   =     14384     ; memory block 2
C000                COLBL2   =     15384     ; color block 1
                                             ;
                                             ; Save the current screen as a help screen on
                                             ; F1. Recall it on F7.
                                             ; Quit on left-arrow key
C000  A9 00                  LDA   #0        ; initialize HELPFL
C002  8D 21 C1               STA   HELPFL
C005  A9 93         CLRCHR   LDA   #147      ; clear the screen
C007  20 D2 FF               JSR   CHROUT
C00A  20 E4 FF      MAINLP   JSR   GETIN     ; get a keypress
C00D  F0 FB                  BEQ   MAINLP    ; if no keypress
C00F  C9 5F                  CMP   #95       ; is it the left-arrow key?
C011  F0 0D                  BEQ   EXIT      ; if so, leave the program
C013  C9 85                  CMP   #133      ; is it F1?
C015  F0 0A                  BEQ   SAVEHS    ; if so, save a help screen
C017  C9 88                  CMP   #136      ; is it F7?
C019  F0 1D                  BEQ   HELP      ; if so, recall a help screen
C01B  20 D2 FF               JSR   CHROUT    ; otherwise, print the character
C01E  D0 EA                  BNE   MAINLP    ; branch always
C020  60            EXIT     RTS             ; exit the program
                                             ;
                                             ; SAVEHS saves a help screen
C021  20 65 C0      SAVEHS   JSR   SWAPCO    ; set zero-page pointers to color memory
                                             ; for two screens
C024  20 8D C0               JSR   SWAPIT    ; swap color memory for the two screens
C027  B0 2E                  BCS   ERROR     ; if color memory overlaps, print error
                                             ; message
C029  20 79 C0               JSR   SWAPSC    ; set zero-page pointers to text for two
                                             ; screens
C02C  20 8D C0               JSR   SWAPIT    ; swap text for the two screens
C02F  B0 26                  BCS   ERROR     ; if screen memory overlaps, print error
                                             ; message and leave
C031  A9 01                  LDA   #1        ; to indicate help screen has been saved
C033  8D 21 C1               STA   HELPFL
C036  D0 CD                  BNE   CLRCHR    ; and continue by clearing screen
                                             ;
                                             ; HELP recalls a help screen.
C038  AD 21 C1      HELP     LDA   HELPFL    ; determine whether a help screen has
                                             ; previously been saved
C03B  F0 CD                  BEQ   MAINLP    ; no help screen has been saved
C03D  20 4B C0               JSR   SWAP2     ; swap in the help screen
C040  20 E4 FF      HELPLP   JSR   GETIN     ; wait for keypress to swap in normal screen
C043  F0 FB                  BEQ   HELPLP    ; if no keypress
C045  20 4B C0               JSR   SWAP2     ; swap in the normal screen
C048  4C 0A C0               JMP   MAINLP    ; and continue
                                             ;
                                             ; Swap primary and help screens.
C04B  20 65 C0      SWAP2    JSR   SWAPCO    ; set zero-page pointers to color memory for
                                             ; two screens
C04E  20 8D C0               JSR   SWAPIT    ; swap color memory for two screens
```

```
C051  20 79 C0            JSR   SWAPSC      ; set zero-page pointers to text for two
                                            ; screens
C054  4C 8D C0            JMP   SWAPIT      ; swap text for two screens and RTS
                                            ;
                                            ; Error message for overlap of two memory
                                            ; blocks.
C057  A0 00      ERROR    LDY   #0          ; as an index
C059  B9 04 C1   ERRLP    LDA   ERRMSG,Y    ; print the message character by character
C05C  F0 06               BEQ   EREXIT      ; exit on a zero byte
C05E  20 D2 FF            JSR   CHROUT      ; print a character
C061  C8                  INY               ; for next character
C062  D0 F5               BNE   ERRLP       ; branch always
C064  60         EREXIT   RTS
                                            ;
                                            ; SWAPCO initializes ZP to screen 1 color
                                            ; and ZP + 2 to screen 2 color.
C065  A9 00      SWAPCO   LDA   #<COLBL1    ; store low and high bytes of screen 1 color
                                            ; to ZP
C067  85 FB               STA   ZP
C069  A0 D8               LDY   #>COLBL1
C06B  84 FC               STY   ZP+1
C06D  A9 18               LDA   #<COLBL2    ; store low and high bytes of screen 2 color to
                                            ; zero page also
C06F  85 FD               STA   ZP+2
C071  A0 3C               LDY   #>COLBL2
C073  84 FE               STY   ZP+3
C075  20 BC C0            JSR   ONELES      ; make sure screen at ZP is lower in memory
                                            ; than the one at ZP+2
C078  60                  RTS
                                            ;
                                            ; SWAPSC initializes ZP to screen 1 text and
                                            ; ZP + 2 to screen 2 text.
C079  A9 00      SWAPSC   LDA   #<BLOCK1    ; store low and high bytes of screen 1 text
                                            ; to ZP
C07B  85 FB               STA   ZP
C07D  A0 04               LDY   #>BLOCK1
C07F  84 FC               STY   ZP+1
C081  A9 30               LDA   #<BLOCK2    ; store low and high bytes of screen 2 text to
                                            ; zero page also
C083  85 FD               STA   ZP+2
C085  A0 38               LDY   #>BLOCK2
C087  84 FE               STY   ZP+3
C089  20 BC C0            JSR   ONELES      ; make sure screen at ZP is lower in memory
                                            ; than the one at ZP+2
C08C  60                  RTS
                                            ;
                                            ; SWAPIT swaps NUMBER bytes at the
                                            ; addresses pointed to by ZP and ZP+2.
C08D  20 B1 C0   SWAPIT   JSR   OVRLAP      ; check for overlapping blocks and store
                                            ; number in COUNTR
C090  90 01               BCC   INITSP      ; memory blocks don't overlap, so continue
C092  60                  RTS               ; memory blocks overlap, so return and
                                            ; print error message
                                            ;
C093  A0 00      INITSP   LDY   #0          ; as an index in SWAPLP
C095  B1 FB      SWAPLP   LDA   (ZP),Y      ; read a byte from first block
                                            ; On the 128, use INDFET in place of the
                                            ; previous instruction
                                            ; to swap memory from bank to bank
                                            ; see MVU128 and MOVEDN for details
C097  AA                  TAX               ; store it in X
C098  B1 FD               LDA   (ZP+2),Y    ; read a byte from second block (if needed,
                                            ; use INDFET on 128)
```

```
C09A  91  FB              STA   (ZP),Y       ; store byte from BLOCK2 into BLOCK1
                                             ; On the 128, use INDSTA in place of the
                                             ; previous instruction
                                             ; to swap memory from bank to bank
                                             ; see MVU128 and MOVEDN for details
C09C  8A                  TXA                ; put byte from BLOCK1 in .A
C09D  91  FD              STA   (ZP+2),Y     ; store byte from BLOCK1 into BLOCK2 (if
                                             ; needed, INDSTA on 128)
C09F  E6  FB              INC   ZP           ; increment low byte of BLOCK1 and
                                             ; BLOCK2
C0A1  D0  02              BNE   INCBL2       ; increment BLOCK2 by 1
C0A3  E6  FC              INC   ZP+1         ; increment high byte of BLOCK1
C0A5  E6  FD      INCBL2  INC   ZP+2         ; increment low byte of BLOCK2
C0A7  D0  02              BNE   LENCHK       ; low byte has yet to turn over, so skip
                                             ; forward
C0A9  E6  FE              INC   ZP+3         ; increment high byte of BLOCK2
C0AB  CE  1D  C1  LENCHK  DEC   COUNTR       ; decrement low byte of counter
C0AE  D0  E5              BNE   SWAPLP       ; if not equal, more remains, so continue
                                             ; swapping bytes
C0B0  CE  1E  C1          DEC   COUNTR+1     ; otherwise, decrement high byte of counter
C0B3  AD  1E  C1          LDA   COUNTR+1     ; keep swapping until last page of buffer
                                             ; has been swapped
C0B6  C9  FF              CMP   #255         ; high byte goes from 0 to 255 on last page
C0B8  D0  DB              BNE   SWAPLP       ; we've yet to reach the last page, so
                                             ; continue switching bytes
C0BA  18                  CLC
C0BB  60                  RTS
                                             ;
                                             ; Make address pointed to by ZP less than
                                             ; address pointed to by ZP+2.
C0BC  A5  FE      ONELES  LDA   ZP+3         ; high byte of screen 2 (text or color)
C0BE  C5  FC              CMP   ZP+1         ; compare with high byte of screen 1 (text
                                             ; or color)
C0C0  F0  03              BEQ   LOWCMP       ; if equal, compare low bytes
C0C2  90  08              BCC   FLIPZP       ; screen at ZP is higher in memory, so flip
                                             ; them
C0C4  60                  RTS                ; no flip necessary based on high bytes
                                             ; alone
C0C5  A5  FD      LOWCMP  LDA   ZP+2         ; low byte of screen 2 (text or color)
C0C7  C5  FB              CMP   ZP           ; compare with low byte of screen 2 (text or
                                             ; color)
C0C9  90  01              BCC   FLIPZP       ; screen at ZP is higher, so flip zero-page
                                             ; pointers
C0CB  60                  RTS                ; no flip necessary
                                             ;
                                             ; Switch ZP pointers, low bytes first.
C0CC  A5  FB      FLIPZP  LDA   ZP           ; get low byte for first screen (text or color)
C0CE  48                  PHA                ; store it on the stack
C0CF  A5  FD              LDA   ZP+2         ; get low byte for second screen (text or
                                             ; color)
C0D1  85  FB              STA   ZP           ; store as low byte for first screen
C0D3  68                  PLA                ; restore low byte for first screen
C0D4  85  FD              STA   ZP+2         ; store as low byte for second screen
C0D6  A5  FC              LDA   ZP+1         ; now do the same for the high bytes
C0D8  48                  PHA
C0D9  A5  FE              LDA   ZP+3
C0DB  85  FC              STA   ZP+1
C0DD  68                  PLA
C0DE  85  FE              STA   ZP+3
C0E0  60                  RTS
                                             ;
                                             ; Determine whether memory blocks
                                             ; overlap and store number of bytes
                                             ; in COUNTR.
```

```
C0E1  AD 1B  C1  OVRLAP   LDA   NUMBER     ; store low byte of number of bytes to swap
C0E4  8D 1D  C1           STA   COUNTR
C0E7  18                  CLC              ; add this to the low byte of the lower
                                           ; block
C0E8  65 FB               ADC   ZP
C0EA  8D 1F  C1           STA   SUM        ; and store low byte result in SUM
C0ED  AA                  TAX              ; save low byte result in .X
C0EE  AD 1C  C1           LDA   NUMBER+1   ; store high byte also
C0F1  8D 1E  C1           STA   COUNTR+1
C0F4  65 FC               ADC   ZP+1       ; add this to the high byte of lower block
C0F6  8D 20  C1           STA   SUM+1      ; and again store high-byte result
C0F9  C5 FE               CMP   ZP+3       ; compare high-byte result with high byte
                                           ; of second block
C0FB  90 06               BCC   NOTOVR     ; if second-block high byte is greater,
                                           ; there's no overlap
C0FD  8A                  TXA              ; otherwise, check the low bytes; get low
                                           ; byte of addition from .X
C0FE  C5 FD               CMP   ZP+2       ; compare with low byte of second block
C100  90 01               BCC   NOTOVR     ; if second-block low byte is greater, there's
                                           ; no overlap
C102  38                  SEC              ; set the carry flag to indicate overlapping
                                           ; memory blocks
C103  60         NOTOVR   RTS
                                           ;
C104  42 4C  4F  ERRMSG   .ASC  "BLOCK 1 AND 2 OVERLAP"
C11A  00                  .BYTE 0          ; terminator byte
C11B  E8 03      NUMBER   .WORD 1000       ; number of bytes to swap
C11D  00 00      COUNTR   .WORD 0          ; counter for the remaining number of bytes
                                           ; to swap
C11F  00 00      SUM      .WORD 0          ; two bytes for sum of BLOCK1 and
                                           ; NUMBER
C121  00         HELPFL   .BYTE 0          ; help screen flag (1 = help screen in
                                           ; memory)
```

See also MOVEDN, MVU128, MVU64.

Name

Switch uppercase to lowercase and vice versa

Description

SWITCH converts the character value in the accumulator to lowercase if it was uppercase, or to uppercase if it was lower case. One application for such a routine is in a word processor program.

Prototype

1. Check the character value to see whether it lies within one of the three valid ranges for alphabetic characters: decimal 193–218, 97–122, or 65–90.
2. If it doesn't, exit the routine, leaving .A intact.
3. If the character in .A is within one of the three ranges, shift left with ASL, moving bit 7 into the carry flag.
4. If carry is clear, the character is either in the range 97–122 or 65–90. In this situation, flip bit 6, changing the case. (Bit 6 will later shift right to become bit 5.) Otherwise, go to step 5 because the character is in the range 193–218.
5. Perform an LSR and then end the routine with RTS.

Explanation

In the example program, a character is fetched from the keyboard. If it's a letter, its case is changed with the subroutine **SWITCH**. The character is then printed and another keypress accepted. To exit the program, press RETURN.

Once it has been established that the accumulator contains a letter between *A* and *Z*, **SWITCH** uses the character's bit pattern to carry out the actual case switching. Take a look at the bit patterns of characters within the three ASCII ranges before and after case switching:

	Before:		**After:**	
	Range	**Bit Pattern**	**Range**	**Bit Pattern**
Lowercase	65–90	%010*x xxxx*	97–122	%011*x xxxx*
Uppercase 1	97–122	%011*x xxxx*	65–90	%010*x xxxx*
Uppercase 2	192–218	%110*x xxxx*	65–90	%010*x xxxx*

Within the bit pattern, a 0 designates bits that are always off, and a 1, bits that are always on. An *x* represents bits that can be on or off.

Converting a character in the range 65–90 to the range 97–122, or vice versa, requires that you flip bit 5. To go from the range 192–218 to 65–90, turn off bit 7.

This is exactly what occurs within FLIPIT. The bits of the letter character are shifted one position to the left with ASL. If the carry flag is set, the character is in the range 192–218. At this point, it's simply a matter of restoring it to its original bit pattern, but with bit 7 off. This is accomplished with LSR, which always shifts a zero into bit 7.

If carry is clear, the character must be in the range 65–90 or 97–122. In this case, bit 6 is flipped (it was previously bit 5), and an LSR is performed, moving bit 6 back to its proper position.

Note: **SWITCH** can easily be modified to narrow the range of characters converted. For instance, to convert only *a*, *b*, and *c* from the lowercase set to uppercase, change RANGE2 to

RANGE2 .BYTE 219,123,68

Also, notice that **SWITCH** uses the Y register. If you access this routine from within a loop indexed by .Y, be sure to save this register to a temporary location first and restore it upon returning.

Routine

```
C000                CHROUT   =    65490
C000                GETIN    =    65508
C000                DSFTCM   =    [illegible]      ; DSFTCM = 11 on the 128
C000                ESFTCM   =    9         ; ESFTCM = 12 on the 128
                                            ;
                                            ; Switch case of input, quit on RETURN.
C000  A9  0E                 LDA  #14       ; set for lowercase mode
C002  20  D2  FF             JSR  CHROUT
C005  A9  08                 LDA  #DSFTCM   ; disable SHIFT/Commodore key
C007  20  D2  FF             JSR  CHROUT
C00A  20  E4  FF    WAIT     JSR  GETIN     ; get a character
C00D  F0  FB                 BEQ  WAIT      ; if no character, then wait
C00F  20  1F  C0             JSR  SWITCH    ; switch case of input
C012  20  D2  FF             JSR  CHROUT    ; print it
C015  C9  0D                 CMP  #13       ; is it RETURN?
C017  D0  F1                 BNE  WAIT      ; no, so get another character
C019  A9  09        QUIT     LDA  #ESFTCM   ; enable SHIFT/Commodore key
C01B  20  D2  FF             JSR  CHROUT
C01E  60                     RTS
                                            ;
                                            ; Switch case of ASCII character in .A.
C01F  A0  03        SWITCH   LDY  #3        ; index to table
C021  88            LOOP     DEY            ; index goes 2-1-0
C022  30  10                 BMI  EXIT      ; if finished checking ranges
C024  D9  35  C0             CMP  RANGE1,Y
C027  90  0B                 BCC  EXIT      ; character is less than RANGE1, so exit
C029  D9  38  C0             CMP  RANGE2,Y
C02C  B0  F3                 BCS  LOOP      ; character is higher than RANGE2, so try
                                            ; next range
C02E  0A            FLIPIT   ASL            ; character is in a range, shift bit 7 into
                                            ; carry
```

```
C02F  B0 02              BCS   FIXIT         ; character is >=128
C031  49 40              EOR   #64           ; flip bit 6
C033  4A        FIXIT    LSR                 ; restore it (bit 7 becomes 0, so 193-218
                                             ; converts to 65-90)
C034  60        EXIT     RTS
                                             ;
C035  C1 61 41  RANGE1   .BYTE 193,97,65
                                             ; lower delimiter of each range
C038  DB 7B 5B  RANGE2   .BYTE 219,123,91
                                             ; upper delimiter+1 of each range
```

See also CNVERT, MIXLOW, MIXUPP.

Name

Convert characters from true ASCII to Commodore ASCII

Description

When you're using a modem to telecommunicate, the characters received over the telephone line will generally be true, or standard, ASCII. Commodore computers use a slightly different character code standard called *Commodore ASCII*. So, any terminal program you write on the 64 or 128 should include a routine like **TASCAS** for converting character codes from true ASCII to Commodore ASCII. Often it will be necessary to perform this character conversion from within a loop indexed by either the X or Y register. Because of this, **TASCAS** was designed to leave both these registers untouched.

Prototype

1. AND the character code value in .A with 127 to insure that it's in the range 0–127.
2. Check the value to see whether it lies within true ASCII uppercase range (65–90).
3. If it's less than 65, then RTS, leaving .A intact.
4. If the value in .A is within the range 65–90, go to step 7.
5. Otherwise, check the character value to see whether it falls within true ASCII lowercase range (97–122).
6. If it's more or less than the range, then RTS, again leaving .A intact.
7. Flip bit 5 and RTS.

Explanation

In the example program, individual bytes representing true ASCII characters are fetched from BUFFER and are then printed; the conversion is done with **TASCAS**, and the resulting Commodore ASCII value is printed. This process continues until a zero byte is read in.

TASCAS takes a true ASCII value in .A and returns an equivalent Commodore ASCII value (also in .A).

Conversion from true ASCII to Commodore ASCII by the routine is a fairly simple matter because of the similarities among the two character sets. True ASCII values lie in a range 0–127. None of the graphics characters present in the upper half of the Commodore set are available in true ASCII.

Both sets are identical in the range 0–127, except for one thing: Uppercase and lowercase letters are reversed. This difference is easily handled within **TASCAS** by flipping bit 5 of

the character value using the EOR command. If you EOR with the number 32, you effectively add (or subtract) 32, depending on whether bit 5 is clear or set.

Routine

```
C000                 CHROUT   =    65490
C000                 LINPRT   =    48589       ; LINPRT = 36402 on the 128
                                               ;
                                               ; Get a number representing a true ASCII
                                               ; character from buffer, and print
                                               ; the number. Convert the character to
                                               ; Commodore ASCII, and print its value.
C000  A0 00                   LDY  #0
C002  B9 45 C0  LOOP          LDA  BUFFER,Y    ; get a true ASCII character
C005  F0 22                   BEQ  QUIT
C007  8D 4D C0                STA  TEMPA       ; save .A
C00A  8C 4E C0                STY  TEMPY       ; save .Y (since LINPRT corrupts .Y)
C00D  20 2A C0                JSR  NUMOUT      ; print the true ASCII value
C010  A9 20                   LDA  #32         ; print SPACE
C012  20 D2 FF                JSR  CHROUT
C015  AD 4D C0                LDA  TEMPA       ; restore .A
C018  20 30 C0                JSR  TASCAS      ; convert .A from true ASCII to Commodore
                                               ; ASCII
C01B  20 2A C0                JSR  NUMOUT      ; print the Commodore ASCII value
C01E  A9 0D                   LDA  #13         ; print RETURN
C020  20 D2 FF                JSR  CHROUT
C023  AC 4E C0                LDY  TEMPY       ; restore .Y
C026  C8                      INY              ; for next value
C027  D0 D9                   BNE  LOOP        ; and get another character
C029  60        QUIT          RTS
                                               ;
C02A  AA        NUMOUT        TAX              ; low byte of true ASCII value (see
                                               ; NUMOUT)
C02B  A9 00                   LDA  #0          ; high byte
C02D  4C CD BD                JMP  LINPRT      ; print the ASCII value
                                               ;
                                               ; Convert true ASCII in .A to Commodore
                                               ; ASCII in .A.
C030  29 7F     TASCAS        AND  #127        ; value must be 0-127
C032  C9 41                   CMP  #65         ; is it less than uppercase A?
C034  90 0E                   BCC  EXIT        ; yes, so leave as is
C036  C9 5B                   CMP  #91         ; is it greater than uppercase Z?
C038  90 08                   BCC  FLIPIT      ; no, so in range 65-90, switch to lowercase.
                                               ; Otherwise, character is in range 91-127.
                                               ; First check for lowercase.
C03A  C9 61     LOWCAS        CMP  #97         ; is it less than lowercase a?
C03C  90 06                   BCC  EXIT        ; yes, so leave it as is
C03E  C9 7B                   CMP  #123        ; is it greater than lowercase z?
C040  B0 02                   [redacted] EXIT  ; yes, so leave as is
                                               ; Character is in lowercase range 97-122, so
                                               ; switch it to uppercase.
C042  49 20     FLIPIT        EOR  #32         ; change uppercase to lowercase or vice
                                               ; versa
C044  60        EXIT          RTS
                                               ;
                                               ; Buffer of true ASCII character bytes.
C045  42 5F 60  BUFFER        .BYTE 66,95,96,33,97,122,90,0
C04D  00        TEMPA         .BYTE0 ; .A storage
C04E  00        TEMPY         .BYTE0 ; .Y storage
```

See also CASSCR, CASTAS, CNVERT, SCRCAS.

Name

Time-of-day (TOD) clock 1 delay

Description

This timer routine is based on the first time-of-day (TOD) clock. **TOD1DL** causes delays within the full range of this clock, from 1/10 second up to 24 hours.

Prototype

1. Before entering this routine, define the delay time in BCD (binary-coded decimal) format as DELAYT in the variables at the end of the program.
2. Using **TOD1ST**, set TOD clock 1 to zero (00:00:00.0 a.m.).
3. Compare the TOD clock 1 reading with the delay specified. Begin with the hours byte, to stop the clock from updating, and work down through the tenths-of-seconds byte.
4. If, before comparing the entire reading, a byte in the clock reading is lower than the corresponding byte in the delay time, read the tenths-of-seconds place to restart the clock and jump to step 3.
5. When a byte from the TOD clock reading exceeds the respective delay-time byte, return from the routine.

Explanation

The example program demonstrates how this routine might be incorporated into your own programs. It prints a message to the screen and allows the user 12 seconds to read it—as timed by **TOD1DL**—before clearing the screen.

One way to achieve the specified delay here would be to add the delay time to the current clock time and then wait for the clock to reach this total. But since the TOD clock keeps time in BCD format, and digits within the clock turn over on different values, this approach would become quite involved. BCD arithmetic counts from 0 through 99, while clocks count from 00 through 59, except the hours (01–12). For example, adding three minutes to 3:58 should result in 4:01, not 3:61.

An easier way to go about this is to start the clock at midnight and then directly compare the delay time with the current TOD time. This is the method used here.

At the outset of **TOD1DL**, each byte within TOD clock 1 is set to zero, beginning with the hours byte. Because of its latching mechanism, the clock doesn't actually start updating until you write to the tenths-of-seconds byte (see **TOD2ST**).

Once all bytes within the clock are set to zero, a byte-by-

byte comparison loop is undertaken. The routine concludes when the clock time exceeds the delay time.

The delay time, DELAYT, is formatted exactly like TIMSET. This allows you to cause delays of up to 24 hours, although we're not sure why you'd ever need such a long delay. But if you do a delay longer than 11 hours, 59 minutes, set the high bit in the hours place when you define DELAYT, just as you would if you were setting a TOD clock (again, see **TOD2ST** for details).

Note: Although based on the first TOD clock, the routine could be modified with little effort to use the second TOD clock. Just replace TODTN1 with TODTN2, and **TOD1ST** with **TOD2ST**, throughout the routine.

Routine

```
C000                 TODTN1  =    56328       ; time-of-day clock 1  tenths-of-seconds
                                              ; register
C000                 TODTN2  =    56584       ; time-of-day clock 2  tenths-of-seconds
                                              ; register
C000                 CHROUT  =    65490
                                              ;
                                              ; Allow 12 seconds to read a message using
                                              ; TOD clock 1 delay
C000  A0 00                  LDY  #0          ; first print a message
C002  B9 4D C0       PRTLOP  LDA  MESSAG,Y    ; get a character from the message string
C005  F0 06                  BEQ  PRTEND      ; quit printing on a zero byte
C007  20 D2 FF               JSR  CHROUT      ; print the character
C00A  C8                     INY              ; for next character
C00B  D0 F5                  BNE  PRTLOP      ; branch always
C00D  20 13 C0       PRTEND  JSR  TOD1DL      ; cause a TOD clock delay
C010  4C 31 C0               JMP  CLRCHR      ; clear the screen and RTS
                                              ;
                                              ; Set up a TOD clock 1 delay.
C013  20 36 C0       TOD1DL  JSR  TOD1ST      ; set TOD clock 1 to all zeros
                                              ; Now wait for current reading to agree
                                              ; with DELAYT.
C016  A0 00          COMPAR  LDY  #0          ; as an index for DELAYT
C018  A2 03                  LDX  #3          ; as an index for hrs., mins., secs., tenths in
                                              ; TOD clock
C01A  BD 08 DC       CMPLOP  LDA  TODTN1,X    ; read TOD clock 1—hrs., mins., secs.,
                                              ; tenths
C01D  D9 49 C0               CMP  DELAYT,Y    ; compare with delay
C020  F0 08                  BEQ  NEXTPL      ; if equal, check the next byte
C022  B0 0C                  BCS  FINIS       ; if TOD byte is greater, time's expired, so
                                              ; return
C024  AD 08 DC               LDA  TODTN1      ; read tenths place to update clock
C027  4C 16 C0               JMP  COMPAR      ; if DELAYT is greater, carry is clear, so
                                              ; continue comparing
C02A  C8             NEXTPL  INY              ; for next DELAYT position
C02B  CA                     DEX              ; for next clock position (mins., secs.,
                                              ; tenths)
C02C  10 EC                  BPL  CMPLOP      ; do all four bytes
C02E  30 E6                  BMI  COMPAR      ; do it all again if time hasn't expired
C030  60             FINIS   RTS              ; we're finished
                                              ;
```

```
C031  A9 93        CLRCHR  LDA   #147          ; time's up, so clear the screen
C033  4C D2 FF             JMP   CHROUT        ; and RTS
                                               ;
                                               ; Set TOD clock 1 (or 2).
                                               ; Replace TODTN1 with TODTN2 to set
                                               ; TOD clock 2.
C036  A0 00        TOD1ST  LDY   #0            ; as an index in TIMSET
C038  A2 03                LDX   #3            ; as an index for hrs., mins., secs., tenths in
                                               ; TODTN1
C03A  B9 45 C0     SETLOP  LDA   TIMSET,Y      ; read in the time to set
C03D  9D 08 DC             STA   TODTN1,X      ; store to clock—hrs. first
C040  C8                   INY                 ; for next byte in TIMSET
C041  CA                   DEX                 ; for next clock byte (mins., secs., tenths)
C042  10 F6                BPL   SETLOP        ; set all four bytes in clock
C044  60                   RTS
                                               ;
C045  00 00 00     TIMSET  .BYTE 0,0,0,0       ; hrs., mins., secs., tenths to set clock
                                               ; (00.00.00.0 a.m.)
C049  00 00 12     DELAYT  .BYTE $0,$0,$12,$0  ; delay in BCD hrs., mins., secs., and tenths
C04D  93 59 4F     MESSAG  .ASC  "{CLR}YOU HAVE 12 SECONDS TO READ THIS."
C06F  00                   .BYTE 0             ; string terminator
```

See also ALARM2, INTCLK, TOD1RD, TOD2PR, TOD2ST, BYT1DL, BYT2DL, INTDEL, JIFDEL, KEYDEL.

Name

Read a time-of-day (TOD) clock

Description

This routine allows you to read either time-of-day clock. It's currently set up to read the first TOD clock, the one in CIA 1. But by substituting TODTN2 for TODTN1 in the routine, the second TOD clock (in CIA 2) can be read. In such instances, **TOD2RD** would be a more appropriate name for the routine.

Prototype

1. Set the Y register, which serves as an index into the buffer holding the current clock reading (BUFFER), to 0. The X register should be initialized to 3 so that the hours place is read first.
2. In RDLOOP, read each byte—either hours, minutes, seconds, or tenths of seconds—from one of the TOD clocks and store it into BUFFER.

Explanation

The TOD clocks have a latching function which prevents them from updating anytime you read or write to them, provided you begin with the hours place and end with the tenths-of-seconds place. This mechanism is described more thoroughly under entry **TOD2ST**, where a TOD clock is set to a specified time.

At any rate, the important point for this routine is that you must read the TOD clock from the hours place to the tenths-of-seconds place. Reading the hours place first stops the clock from updating. Only when you read (or write to) the tenths-of-seconds place will the clock continue updating.

The time read in from a TOD clock, whether it's clock 1 or 2, is in a binary-coded decimal format. This reading is stored here in BUFFER as a four-byte number, just as it appears in the clock. Each half-byte, or hexadecimal digit, actually represents a decimal digit in the clock reading.

For example, if the clock reading in BUFFER were $91,$49,$32,$04, the time would be 11:49:32.4 p.m. (The high bit in the hours byte serves as an a.m./p.m. flag.)

Routine

```
C000                TODTN1   =     56328        ; time-of-day clock 1—tenths-of-seconds
                                                ; register
C000                TODTN2   =     56584        ; time-of-day clock 2—tenths-of-seconds
                                                ; register
                                                ;
                                                ; Read TOD clock 1 (or 2) and store the
                                                ; reading to a memory buffer
                                                ; Replace TODTN1 with TODTN2 to read in
                                                ; TOD clock 2.
C000  A0 00         TOD1RD   LDY   #0           ; as an index for buffer position
C002  A2 03                  LDX   #3           ; as an index for hrs., mins., secs., tenths
C004  BD 08 DC      RDLOOP   LDA   TODTN1,X     ; read the TOD clock—hrs., mins., secs.,
                                                ; tenths
C007  99 0F C0               STA   BUFFER,Y     ; store to buffer
C00A  C8                     INY                ; for next buffer position
C00B  CA                     DEX                ; for next clock position (mins., secs.,
                                                ; tenths)
C00C  10 F6                  BPL   RDLOOP       ; read four bytes
C00E  60                     RTS
                                                ;
C00F  00 00 00      BUFFER   .BYTE 0,0,0,0      ; Storage for clock reading. Stored in BCD
                                                ; format as
                                                ; hrs., mins., secs., and tenths.
```

See also ALARM2, INTCLK, TOD1DL, TOD2PR, TOD2ST.

Name

Print the time-of-day (TOD) time

Description

TOD2PR prints the current reading for time-of-day clock 2 in the upper left corner of the screen. As with the other TOD clock routines presented in the book, the remaining TOD clock can be used instead. In this case, simply replace TODTN2 in the routine with TODTN1. If you like, you can also change the name of the routine to **TOD1PR** to indicate that TOD clock 1 is being printed.

Prototype

1. Set the Y register, which serves to index the screen position, to zero. The X register is initialized to 3 so that the hours byte is read first.
2. In PRTLOP, read a byte—either hours, minutes, seconds, or tenths of seconds—from one of the two TOD clocks.
3. Shift the high nybble of this byte into its low nybble, convert this to a numeric screen code, and store it in screen memory.
4. Mask out the high nybble of the byte taken in Step 2. Convert the remaining low nybble to a screen code and store it to the screen.
5. For the tenths-of-seconds byte, only the low nybble is displayed.
6. After each half byte from the TOD clock has been positioned on the screen in Steps 3, 4, and 5, store the screen code for a colon (or for a decimal following the seconds place).
7. When PRTLOP finishes, skip a space on the screen and store either the screen code for P (representing p.m.) or A (for a.m.) in screen memory depending on the setting of bit 7 of the hours byte. Then return from the routine.

Explanation

The program below clears the screen, then jumps to **TOD2PR** to display the current time setting in the second TOD clock.

Each TOD clock, whether it's clock 1 or 2, ceases to update as soon as the hours byte is read (or written to). It continues updating only when the tenths-of-seconds byte is accessed. (See **TOD2ST** for details on this latching function.) For this reason, you should always read these clocks from the hours place down, as we've done here.

The TOD clocks keep time in binary-coded format, making conversion of the clocks' registers to screen codes relatively easy. In **TOD2PR**, bytes from TOD clock 2's registers are separated into half-bytes, which are in turn converted to screen codes and displayed.

To make the display more readable, a colon is placed between the digit pairs representing the hours, minutes, and seconds place. A decimal point follows the seconds place. After all digits from the TOD readout are displayed on the screen, either *A* or *P* (for a.m. or p.m.) is printed.

Routine

```
C000                 TODTN2  =    56584        ; time-of-day clock 2—tenths-of-seconds
                                               ; register
C000                 TODTN1  =    56328        ; time-of-day clock 1—tenths-of-seconds
                                               ; register
C000                 CHROUT  =    65490
C000                 SCREEN  =    1024         ; first text-screen position
                                               ;
                                               ; Clear the screen, read and print TOD clock
                                               ; 2 (or 1).
                                               ; Replace TODTN2 with TODTN1 to read
                                               ; and print TOD clock 1.
C000  A9 93          CLRCHR  LDA  #147         ; clear the screen
C002  20 D2 FF               JSR  CHROUT
C005  4C 08 C0               JMP  TOD2PR       ; print TOD clock 2 and RTS
                                               ;
                                               ; Read and print TOD clock 2.
C008  A0 00          TOD2PR  LDY  #0           ; initialize index to screen position
C00A  A2 03                  LDX  #3           ; initialize index for hrs., mins., secs., and
                                               ; tenths
C00C  BD 08 DD       PRTLOP  LDA  TODTN2,X     ; read the TOD clock—hrs., min., sec.,
                                               ; tenths
C00F  E0 00                  CPX  #0           ; skip tenths high nybble
C011  F0 10                  BEQ  LOWNIB
C013  48                     PHA               ; store it temporarily
C014  29 70                  AND  #%01110000   ; mask out low nybble and bit 7
C016  4A                     LSR               ; shift high nybble into low nybble
C017  4A                     LSR
C018  4A                     LSR
C019  4A                     LSR
C01A  09 30                  ORA  #48          ; effectively add 48 to put in numeric range
C01C  99 00 04               STA  SCREEN,Y     ; POKE it to the screen
C01F  C8                     INY               ; next screen position
C020  68                     PLA               ; restore the byte and get second digit from
                                               ; low nybble
C021  29 0F                  AND  #$0F         ; mask out high nybble
C023  09 30          LOWNIB  ORA  #48          ; add 48
C025  99 00 04               STA  SCREEN,Y     ; POKE low nybble's digit to the screen
C028  C8                     INY               ; next screen position
C029  E0 01                  CPX  #1           ; we want to put a decimal between
                                               ; seconds and tenths
C02B  F0 04                  BEQ  POINT        ; POKE a decimal point
C02D  90 0F                  BCC  NEXTPL       ; don't print the last colon
C02F  D0 07                  BNE  COLON        ; we're not between seconds and tenths
C031  A9 2E          POINT   LDA  #46          ; screen code for decimal point
C033  99 00 04               STA  SCREEN,Y     ; POKE a decimal point
C036  D0 05                  BNE  CONTLP       ; branch always
```

```
C038  A9 3A      COLON   LDA  #58        ; POKE a colon between hrs., mins., and
                                         ; secs.
C03A  99 00 04           STA  SCREEN,Y
C03D  C8         CONTLP  INY             ; next screen position
C03E  CA         NEXTPL  DEX             ; for next clock position (min., sec., tenths)
C03F  10 CB              BPL  PRTLOP     ; read and print four bytes
C041  C8                 INY             ; skip a space
C042  AD 0B DD           LDA  TODTN2+3   ; get the hours byte
C045  30 06              BMI  PMFLAG     ; bit 7 is set indicating p.m.
C047  A9 01              LDA  #1         ; screen code for A (a.m.)
C049  99 00 04   PRAMPM  STA  SCREEN,Y   ; POKE a.m./p.m. flag to screen
C04C  60                 RTS
C04D  A9 10      PMFLAG  LDA  #16        ; screen code for P (p.m.)
C04F  D0 F8              BNE  PRAMPM     ; print it
```

See also ALARM2, INTCLK, TOD1DL, TOD1RD, TOD2ST.

Name

Set a time-of-day (TOD) clock

Description

Each of the two CIA (complex interface adapter) chips in the 64 and 128 has a built-in time-of-day (TOD) clock. Unlike the jiffy clock, which is maintained via software (the IRQ interrupt service routine), the TOD clocks are updated automatically by CIA hardware. The TOD clocks aren't used at all by the operating system, and neither the 64 or 128 provide any facilities in ROM for reading or setting the TOD clocks.

With this routine, you can set either time-of-day clock. As it's currently written, the routine sets the second TOD clock (the clock in CIA #2). But you can just as easily have it set the clock in CIA #1 by replacing TODTN2 with TODTN1 within the routine. In fact, this has been done elsewhere in the book. See entries **INTCLK** and **TOD1DL**. In those instances, this routine is referred to as **TOD1ST**.

Prototype

1. Initialize .Y to 0 and .X to 3. (The Y register indexes the buffer containing the actual time to be set, or TIMSET, at the end of the routine. The offset into the TOD clock is .X.)
2. In a loop, read the four bytes containing the time setting and store them to a TOD clock.

Explanation

When you set either TOD clock, you must begin with the hours place. This is because the TOD clocks have a built-in latching function. Each clock stops updating as soon as you read or write to the hours place and doesn't start again until you write to the tenths-of-seconds place. (The internal registers for either clock, where the actual time is kept, are maintained during this process.) This approach prevents the TOD clock from advancing while you're in the middle of reading or setting it.

The TOD clocks keep time in a binary-coded decimal format. Each hexadecimal digit, or half byte, in the clocks' registers is interpreted as a decimal digit. So, the example time listed in TIMSET as $06,$59,$59,$0 is 59 minutes and 59 seconds after six o'clock. In this case, the time is a.m. The high bit in the hours byte serves as an a.m./p.m. flag. To set the clock to a p.m. time, simply add $80 to the hours byte.

In this routine, writing to the TOD registers sets the cur-

rent time. But these registers can also be used to store an alarm time if the TOD clock is used as an alarm clock. Bit 7 of CIA control register B is the key (CI2CRB at 56591 for TOD clock 2 or CIACRB at 56335 for TOD clock 1). Normally, this bit is zero. But, if you set it to one, the time assigned to the TOD registers is taken as an alarm time. Routine **ALARM2** demonstrates this technique.

Note: The TOD clocks have a bug in the a.m./p.m. function. The normal way to count time is to consider noon to be 12:00 p.m. and midnight to be 12:00 a.m. Thus, the p.m. hours count from 12 to 1 to 2 to 3, and so on, up to 11. But the CIA chip counts p.m. hours from 1 to 12 (which seems more logical, although it's not how things are done in the real world).

If you set the TOD hours byte to 12, on the next hour, the a.m./p.m. flag bit will reverse state. For example, if you set the clock to noon (12:00 p.m.), it will read 1:00 a.m. when the clock reaches 1:00 in the afternoon (1:00 p.m.).

You can get around this problem, though. If the hours place is to be set to 12, just flip the a.m./p.m. flag bit before setting the clock. So, 12:15:16.0 a.m. would be entered in TIMSET as .BYTE $82,$15,$16,$0.

Routine

```
C000                TODTN2   =      56584          ; time-of-day clock 2—tenths-of-seconds
                                                   ; register
C000                TODTN1   =      56328          ; time-of-day clock 1—tenths-of-seconds
                                                   ; register
                                                   ;
                                                   ; Set TOD clock 2 (or 1).
                                                   ; Replace TODTN2 with TODTN1 to set TOD
                                                   ; clock 1
C000  A0  00        TOD2ST   LDY    #0             ; as an index in TIMSET
C002  A2  03                 LDX    #3             ; as an index for hrs., mins., secs., tenths of
                                                   ; secs. in TODTN2
C004  B9  0F  C0    SETLOP   LDA    TIMSET,Y       ; read in the time to set
C007  9D  08  DD             STA    TODTN2,X       ; store to clock—hrs. first
C00A  C8                     INY                   ; for next TIMSET byte
C00B  CA                     DEX                   ; for next clock byte (min., sec., tenths of
                                                   ; secs.)
C00C  10  F6                 BPL    SETLOP         ; set all four bytes of clock
C00E  60                     RTS
                                                   ;
C00F  06  59  59    TIMSET   .BYTE  $06,$59,$59,$0
                                                   ; hr., min., sec., tenths to set clock
                                                   ; (06.59.59.0 a.m.)
                                                   ; For p.m., add in $80 to hour setting.
```

See also ALARM2, INTCLK, TOD1DL, TOD1RD, TOD2PR.

Name

Set the text color using CHR$

Description

TXTCCH outputs the appropriate ASCII color value with CHROUT. This approach is often more convenient than storing a color value in the text color register. Text colors can easily be switched from within an ASCII string definition, as the example illustrates.

Prototype

1. Set up a string containing certain ASCII color codes at the end of your program.
2. JSR to a string printing routine and RTS (or simply JMP to it).

Explanation

Each character of the message HELLO is printed in a different color using **STRCPT**.

Routine

```
C000                CHROUT  =      65490
C000                ZP      =      251
                                          ;
                                          ; Print each character of the string HELLO in
                                          ; a different color.
C000  20 04 C0 TXTCCH  JSR    STRCPT      ; print the string
C003  60               RTS
                                          ;
                                          ; Custom string printing routine
C004  A9 1E    STRCPT  LDA    #<STRING    ; low byte of string
C006  85 FB            STA    ZP          ; store it
C008  A9 C0            LDA    #>STRING    ; high byte of string
C00A  85 FC            STA    ZP+1        ; store it also
C00C  A0 00            LDY    #0          ; as an index
C00E  B1 FB    STRLOP  LDA    (ZP),Y      ; load each character from string
C010  F0 0B            BEQ    FINISH      ; if zero byte, then finished
C012  20 D2 FF         JSR    CHROUT      ; print character
C015  C8               INY                ; for next character
C016  D0 F6            BNE    STRLOP      ; if not more than 256 bytes, then get the
                                          ; next character
C018  E6 FC            INC    ZP+1        ; otherwise, increment high byte address
                                          ; pointer to the string
C01A  4C 0E C0         JMP    STRLOP      ; and continue printing
C01D  60       FINISH  RTS
                                          ;
C01E  05 48 9C STRING  .ASC   "{WHT}H{PUR}E{YEL}L{BLK}L{LT BLU}O"
                                          ; "HELLO" in colors
C028  00               .BYTE  0           ; ending in a zero byte
```

See also BCKCOL, BORCOL, COLFIL, TXTCOL.

Name

Input a line of text using a custom routine

Description

TXTCIN simulates the BASIC ROM routine INLIN for accepting a line of input from the keyboard, blinking cursor and all. But unlike INLIN, which takes an entire line of input at once, **TXTCIN** screens each character individually before adding it to the input line. By building the input line in this manner, the many documented problems associated with INLIN (or INPUT) can be avoided. Thus, commas and characters like the cursor keys, CLEAR, HOME, and so on, can be handled appropriately by the input routine.

Prototype

1. Enable the cursor.
2. Get a character with GETIN.
3. Compare the input character with a table of unwanted characters (BADKEY).
4. If the character is found in the table of unacceptable characters, go to step 2.
5. If the character is DELETE, see whether we're at the start of the buffer. If so, go to step 2. Otherwise, decrement the buffer index (.Y) by 2.
6. If the character is RETURN, print it while the cursor is off, add a zero byte to the buffer, and RTS.
7. If the input character is not RETURN, see whether the input line has reached its maximum length (MAXLEN). If it has, wait for a RETURN.
8. Otherwise, add the character to the input buffer, increment the buffer index .Y, print the character (again, while the cursor is off), and go to step 2 for another character.

Explanation

The main routine in the example is exactly like the one shown for **TXTINP**. A line of input is first retrieved, in this case by **TXTCIN**, and the resulting string data in the input buffer printed with a modified **STRCPT**. (**STRCPT** is shortened since the string is fewer than 256 bytes long.) As with **TXTINP**, we return to BASIC by jumping through the error handler vector at 768.

With a few changes, the input routine **TXTCIN** can be customized for each input required in your program. First, POKE MAXLEN with the maximum number of characters al-

lowed in the current input line. Then, update the table of unwanted keys (BADKEY) and total the number of these keys. POKE this number, less 1, into the location corresponding to NUMBAD ($C026, in this example).

Notice how the cursor is dealt with within the routine. IRQ interrupts must be disabled before each input character is printed and reenabled afterward. Otherwise, the cursor may flash during normal interrupt handling. If this happens, the character will appear on the screen in reverse video.

Cursor handling within **TXTCIN** is certainly tedious and adds a number of bytes to the routine. If a cursor is not required in your program, you can eliminate all instructions necessary to set it up and shorten the routine considerably.

Note: The use of the vector at 768 to exit the routine is required here to prevent BASIC from taking your input as a direct command. See **TXTINP** for more discussion of this.

Routine

```
C000                CHROUT  =     65490
C000                GETIN   =     65508
C000                BUF     =     512
C000                ZP      =     251
C000                YSAVE   =     253
C000                BLNSW   =     204          ; BLNSW = 2599 on the 128
C000                BLNCT   =     205          ; BLNCT = 2600 on the 128
C000                BLNON   =     207          ; BLNON = 2598 on the 128
                                               ;
                                               ; Input a line of text with a custom routine
                                               ; and print it.
C000  20 1C C0              JSR   TXTCIN       ; get the input line
                                               ; Print it with shortened STRCPT and return.
C003  A9 00         STRCPT  LDA   #<BUF        ; low byte of input buffer
C005  85 FB                 STA   ZP           ; store it
C007  A0 02                 LDY   #>BUF        ; high byte of input buffer
C009  84 FC                 STY   ZP+1         ; store it also
C00B  A0 00                 LDY   #0           ; as an index
C00D  B1 FB         STRLOP  LDA   (ZP),Y       ; load each character from input buffer
C00F  F0 06                 BEQ   FINISH       ; if zero byte, then finished
C011  20 D2 FF              JSR   CHROUT       ; print character
C014  C8                    INY                ; next character
C015  D0 F6                 BNE   STRLOP       ; go get next character
C017  A2 80         FINISH  LDX   #128         ; code for READY error message
C019  6C 00 03              JMP   (768)        ; return to BASIC and print READY prompt
                                               ;
                                               ; Custom input subroutine using GETIN and
                                               ; flashing cursor
C01C  A0 00         TXTCIN  LDY   #0           ; initialize index into input buffer
C01E  84 CC                 STY   BLNSW        ; turn on cursor
C020  84 FD         GETKEY  STY   YSAVE        ; GETIN corrupts .Y, so save it
C022  20 E4 FF      WAIT    JSR   GETIN        ; get a character in .A
C025  A2 07                 LDX   #NUMBAD
C027  DD 6B C0      CKLOOP  CMP   BADKEY,X     ; compare character to each value in
                                               ; BADKEY table
C02A  F0 F6                 BEQ   WAIT         ; if response is illegal, get another key
C02C  CA                    DEX
```

```
C02D  10  F8           BPL   CKLOOP      ; check next bad key
C02F  A4  FD           LDY   YSAVE       ; input is okay, so restore .Y
C031  C9  14           CMP   #20         ; is it DELete?
C033  D0  06           BNE   NOTDEL      ; not DELete
C035  C0  00           CPY   #0          ; are we at the start of the buffer?
C037  F0  E7           BEQ   GETKEY      ; If so, go get a character
C039  88               DEY               ; if DELete, back up index into input buffer
C03A  88               DEY
C03B  C9  0D    NOTDEL CMP   #13         ; is it RETURN?
C03D  F0  09           BEQ   PRTIT       ; yes, so print it
C03F  CC  73  C0       CPY   MAXLEN      ; check maximum input length
C042  F0  DC           BEQ   GETKEY      ; if yes, wait for RETURN
C044  99  00  02       STA   BUF,Y       ; store character in buffer
C047  C8               INY               ; increment input buffer index
C048  A2  01    PRTIT  LDX   #1          ; routine to print each character
C04A  86  CD           STX   BLNCT       ; set cursor timer
C04C  A6  CF    WAITPR LDX   BLNON
C04E  D0  FC           BNE   WAITPR      ; wait till flash is off
C050  78               SEI               ; turn off all IRQ interrupts so cursor won't
                                         ; flash
C051  20  D2  FF       JSR   CHROUT      ; print the character
C054  58               CLI               ; turn on IRQ interrupts
C055  C9  0D           CMP   #13         ; is it RETURN?
C057  D0  C7           BNE   GETKEY      ; get another key if not RETURN
C059  A9  00           LDA   #0
C05B  99  00  02       STA   BUF,Y       ; If RETURN, add terminator byte of zero
                                         ; to the string
C05E  A9  01           LDA   #1
C060  85  CD           STA   BLNCT       ; make cursor flash
C062  A5  CF    WAITBL LDA   BLNON
C064  D0  FC           BNE   WAITBL      ; wait until cursor not flashed
C066  A9  01           LDA   #1
C068  85  CC           STA   BLNSW       ; turn off cursor
C06A  60               RTS
                                         ;
C06B  00        BADKEY .BYT  0           ; if no key, then wait
C06C  91  11  9D       .ASC  "{UP}{DOWN}{LEFT}{RIGHT}"
                                         ; cursor keys
C070  94               .ASC  "{INST}"    ; INST key
C071  13  93           .ASC  "{HOME}{CLR}"
                                         ; HOME and CLR
073             NUMBAD =     * BADKEY 1
C073  0A        MAXLEN .BYTE 10          ; maximum length of the input line
```

See also TXTINP.

Name

Set the text color

Description

TXTCOL sets the text color by storing the appropriate color value in the text color flag at location 646 (location 241 on the 128).

Prototype

1. Enter this routine with the selected color value in .A.
2. Store this value in the foreground color register for text (COLOR).

Explanation

The example program makes the text that follows green in color. See **COLFIL** for a table of color values.

Routine

```
C000                COLOR   =     646       ; COLOR = 241 on the 128—foreground
                                            ; color for text
                                            ;
                                            ; Set text color to green
C000  AD 0B  C0             LDA   COLVAL    ; get the color value
C003  20 07  C0             JSR   TXTCOL    ; and set it
C006  60                    RTS
                                            ;
                                            ; Set text color. Enter with .A containing color
                                            ; value.
C007  8D 86  02   TXTCOL    STA   COLOR     ; set text color
C00A  60                    RTS
                                            ;
                  COLVAL    .BYTE 5         ; color green
```

See also BCKCOL, BORCOL, COLFIL, TXTCCH.

Name

Input a line of text using the ROM routine INLIN

Description

You'll find this short routine practical in many programs. **TXTINP** accepts a line of input from the keyboard and stores it as a zero-terminated string in the input buffer beginning at location 512.

Prototype

Jump to the BASIC ROM subroutine INLIN.

Explanation

TXTINP relies on the built-in BASIC Kernal routine, INLIN, to perform an INPUT in ML. INLIN, located at 42336 on the 64 or 22176 on the 128, accepts characters from the current input device until a carriage return is received or until the length of the current logical line is exceeded (80 characters on the 64; 160 on the 128). If the input carries you to the next logical line, that line will become the input line, just as in BASIC. Once you have entered RETURN, INLIN tags a zero byte onto the end of the input line in the buffer.

In the example, **TXTINP** fetches characters from the keyboard, placing them in the text input buffer at 512 until RETURN is pressed. A shortened **STRCPT** is used to print this string data (shortened because the string will never be longer than 255 bytes). After this, you're returned to BASIC.

Notice that instead of using RTS to return to BASIC, we jump through the vector at 768 to BASIC's error message handler routine. (A value of 128 in the X register indexes the READY prompt from a table of error messages.) This is necessary here since BASIC's input buffer has been corrupted with input from INLIN. You'll see what we mean if you substitute an RTS for LDX #128:JMP (768). BASIC will attempt to execute whatever input follows on the current line as if it were a direct command.

Note: Since **TXTINP** uses BASIC's own INPUT routine, it suffers from all the problems inherent to this statement. Punctuation characters like commas and colons cannot be entered within the input line; control characters like the cursor keys,

CLEAR, and HOME allow the user to leave the input line; and so on. Such input can have disastrous effects upon your program. In many instances, especially where the user is likely to be a novice, you should use a custom routine like **TXTCIN**, which screens individual characters within the input line.

Routine

```
C000                CHROUT   =     65490
C000                BUF      =     512
C000                ZP       =     251
C000                INLIN    =     42336      ; INLIN = 22176 on the 128
                                              ;
                                              ; Input a line of text until RETURN and
                                              ; print it.
C000  20  1C  C0             JSR   TXTINP     ; input a line of text into keyboard buffer
                                              ;
                                              ; Now print it with a shortened version of
                                              ; STRCPT (buffer is <256 bytes)
C003  A9  00        STRCPT   LDA   #<BUF      ; low byte of input buffer
C005  85  FB                 STA   ZP         ; store it
C007  A0  02                 LDY   #>BUF      ; high byte of input buffer
C009  84  FC                 STY   ZP+1       ; store it also
C00B  A0  00                 LDY   #0         ; as an index
C00D  B1  FB        STRLOP   LDA   (ZP),Y     ; load each character from input buffer
C00F  F0  06                 BEQ   FINISH     ; if zero byte, then finished
C011  20  D2  FF             JSR   CHROUT     ; print character
C014  C8                     INY              ; for next character
C015  D0  F6                 BNE   STRLOP     ; go get the next character
C017  A2  80        FINISH   LDX   #128       ; code for READY error message
C019  6C  00  03             JMP   (768)      ; return to BASIC and print READY prompt
                                              ;
                                              ; Input a line of text into the keyboard buffer
                                              ; with the BASIC ROM routine INLIN.
C01C  20  60  A5    TXTINP   JSR   INLIN
C01F  60                     RTS
```

See also TXTCIN.

Name

Validate a disk

Description

This is the equivalent of the BASIC statement OPEN 1,8,15,"V0":CLOSE 1, which reads through the directory and checks the allocation of disk sectors. There's no need to validate very often, though if you accidentally leave a disk file open when you turn off the computer, the result is a *poison,* or *splat,* file, which may cause significant problems in the future. You should not scratch a splat file, which is marked in the directory with an asterisk (*) next to the file type; you should validate the disk that contains the poison file.

Prototype

1. Open the command channel (Kernal SETLFS, SETNAM, OPEN).
2. Provide "V0" as the name of the file being opened.
3. Close the channel.

Explanation

At the start of the routine, SETLFS sets a logical file number 1, on device 8 (the disk drive) and channel 15. SETNAM sets the name to "V0", which means *Validate on drive 0.* The Kernal OPEN routine is sufficient to send this command to the disk drive. To finish up, close the channel.

The validate normally takes some time to finish. This is because it reads through the directory to find every legitimate file, then traces through the sectors each program or file uses. Each valid sector is then marked as already used in the block allocation map (BAM).

Warning: Do *not* use the validate routine if you have a double-sided 1571 disk in the drive, and the 1571 is in single-sided 1541 mode. You'll lose the second half of the disk. To be safe, send the double-sided (1571 mode) command "U0>M1" to the disk drive on channel 15 before you validate the disk.

You should also avoid using this routine to validate disks formatted for use with the new *GEOS* operating system for the 64. *GEOS* provides its own Validate program. Performing a standard validation on a *GEOS* disk will result in the loss of vital information.

Routine

```
C000                 SETLFS    =     $FFBA
C000                 SETNAM    =     $FFBD
C000                 OPEN      =     $FFC0
C000                 CLOSE     =     $FFC3
C000                 CLRCHN    =     $FFCC
                                                  ;
C000  A9 01          VALIDT    LDA   #1           ; logical file number
C002  A2 08                    LDX   #8           ; device number for disk drive
C004  A0 0F                    LDY   #15          ; secondary address for drive command
                                                  ; channel
C006  20 BA FF                 JSR   SETLFS       ; prepare to open it
C009  A9 03                    LDA   #BUFLEN      ; length of buffer
C00B  A2 1E                    LDX   #<BUFFER     ; .X and .Y hold the
C00D  A0 C0                    LDY   #>BUFFER     ; address of the buffer
C00F  20 BD FF                 JSR   SETNAM       ; set name
C012  20 C0 FF                 JSR   OPEN         ; open it
C015  A9 01                    LDA   #1           ; and immediately
C017  20 C3 FF                 JSR   CLOSE        ; close the command channel
C01A  20 CC FF                 JSR   CLRCHN       ; clear the channels
C01D  60                       RTS                ; all done
                                                  ;
                                                  ; Data area
C01E  56 30          BUFFER    .ASC  "V0"
C020  0D                       .BYTE 13           ; RETURN character
C021                 BUFLEN    =     * - BUFFER
```

See also CONCAT, COPYFL, FORMAT, INITLZ, RENAME, SCRTCH.

Name

Write to 80-column attribute memory

Description

If you've worked with the 40-column screen of the 64 or 128, you're probably used to color memory that can hold 16 different values. The 128's 80-column screen has attribute memory that not only controls colors, but also controls flash mode, underline mode, reverse mode, lowercase/uppercase or uppercase/graphics mode, and so forth. This routine changes the attributes of a chunk of the screen.

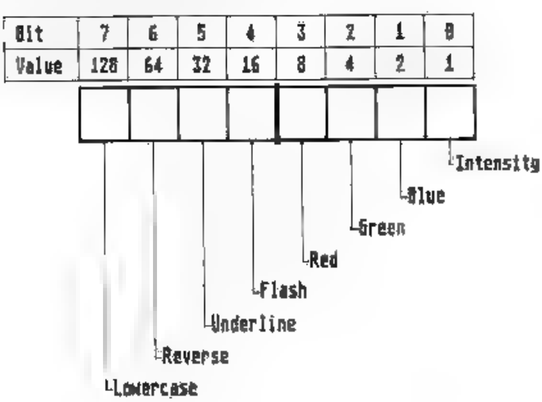
Prototype

1. Enter the routine with the attribute value in .A and the screen position in .X and .Y.
2. Save the attribute temporarily.
3. Calculate the color address from .X and .Y.
4. Send the corresponding address for attribute memory to the VDC chip.
5. Store the attribute into attribute memory.

Explanation

The 128's 80-column screen has 80 columns and 25 rows, a total of 2000 locations. Within its private 16K of memory, there are 2000 bytes devoted to screen memory, plus 2000 bytes for attribute memory. The figure shows how an individual byte of attribute memory controls the functions.

Attribute Memory Byte



The low nybble (bits 0–3) controls the color, with various combinations of red, green, blue, and intensity. The high nybble (bits 4–7) controls the additional attributes such as flash, underline, reverse, and lowercase. For example, if the underline bit is a 1, the character is underlined. If the lowercase bit is 1, the letter *A* appears as lowercase *a*. (If it's 0, an *A* appears as an uppercase *A*, and uppercase letters print as graphics characters.)

The example program stores a %10111101 into attribute memory at *x* position 9, *y* position 4—column 10, row 5, because the upper left corner is (0,0). It stores the value into ten bytes. The upper nybble of %1011 turns on lowercase, underline, and flash. The lower nybble of %1101 turns the color to bright yellow (green + red + intensity).

For more about how the internal VDC registers work, see **RE80CO** and **WR80CO**.

Routine

```
0C00                CHROUT  =    $FFD2         ; Kernal print routine
0C00                VDCADR  =    54784         ; gateway byte 1—the register address
0C00                VDCDAT  =    54785         ; gateway byte 2—the data to be written
0C00                VRMHI   =    18            ; register for memory address to access (high
                                               ; byte)
0C00                VRMLO   =    19            ; register for memory address (low byte)
0C00                VRDAT   =    31            ; register for number to be sent
0C00                COLMEM  =    $0800         ; address of color memory in the VDC's
                                               ; private memory
                                               ;
0C00  A9 93                 LDA  #147          ; clear screen
0C02  20 D2 FF              JSR  CHROUT
0C05  A2 07                 LDX  #>1999
0C07  A0 CF                 LDY  #<1999
0C09  A9 45                 LDA  #69           ; the letter E
0C0B  20 D2 FF  LP01        JSR  CHROUT        ; print it
0C0E  88                    DEY
0C0F  D0 FA                 BNE  LP01
0C11  CA                    DEX
0C12  D0 F7                 BNE  LP01          ; 1999 times
0C14  A9 BD                 LDA  #%10111101    ; lowercase, underline, flash, bright yellow
0C16  A2 09                 LDX  #9            ; x position 9
0C18  A0 04                 LDY  #4            ; y position 4
0C1A  20 28 0C              JSR  VDCCOL        ; store it
0C1D  A0 0A                 LDY  #10           , ten more times
0C1F  A2 1F     SVCLP       LDX  #31           ; register 31
0C21  20 99 0C              JSR  STRVDC        ; store it
0C24  [redacted]            DEY
0C25  D0 F8                 BNE  SVCLP         ; and branch back ten times
0C27  60                    RTS
                                               ;
                                               ; Enter VDCCOL with the number to be
                                               ; POKEd to color/attribute memory in .A,
                                               ; and the x and y locations in .X and .Y.
0C28  8D 96 0C  VDCCOL      STA  TEMPA         ; save .A
0C2B  A9 00                 LDA  #0            ; clear the address
0C2D  8D 97 0C              STA  COLADR        ; of color memory low
```

```
0C30  8D 98 0C          STA   COLADR+1   ; and high byte
0C33  98                TYA              ; move .Y to .A
0C34  0A                ASL              ; times 2
0C35  8D 97 0C          STA   COLADR     ; save it
0C38  0E 97 0C          ASL   COLADR     ; times 4 (low)
0C3B  2E 98 0C          ROL   COLADR+1   ; (high)
0C3E  0E 97 0C          ASL   COLADR     ; times 9 (low)
0C41  2E 98 0C          ROL   COLADR+1   ; (high)
0C44  18                CLC              ; now add in .A
0C45  6D 97 0C          ADC   COLADR     ; times 8 plus times 2 is times 10 (net)
0C48  8D 97 0C          STA   COLADR     ; store it
0C4B  A9 00             LDA   #0         ;
0C4D  6D 98 0C          ADC   COLADR+1   ; fix the high byte
0C50  8D 98 0C          STA   COLADR+1   ; and store it
0C53  A0 03             LDY   #3         ; times 10 times 8 (again) is times 80
0C55  0E 97 0C  LOOP80  ASL   COLADR
0C58  2E 98 0C          ROL   COLADR+1
0C5B  88                DEY
0C5C  D0 F7             BNE   LOOP80
                                         ;
                                         ; Now COLADR holds 0, 80, 160, and so
                                         ; forth.
0C5E  8A                TXA              ; put .X in .A and
0C5F  6D 97 0C          ADC   COLADR     ; add it (carry is always clear)
0C62  8D 97 0C          STA   COLADR     ; store it
0C65  A9 00             LDA   #0         ; high byte, too
0C67  6D 98 0C          ADC   COLADR+1
0C6A  8D 98 0C          STA   COLADR+1
                                         ; Now COLADR holds a number 0-1999,
                                         ; for the screen/color memory location.
0C6D  A9 00             LDA   #<COLMEM   ; add in the beginning of color memory
                                         ; ($0800 inside the VDC)
0C6F  6D 97 0C          ADC   COLADR
0C72  8D 97 0C          STA   COLADR
0C75  A9 08             LDA   #>COLMEM
0C77  6D 98 0C          ADC   COLADR+1
0C7A  8D 98 0C          STA   COLADR+1
0C7D  A2 12             LDX   #VRMHI     ; set the high byte
0C7F  AD 98 0C          LDA   COLADR+1   ; to point to color memory
0C82  20 99 0C          JSR   STRVDC     ; store it
0C85  A2 13             LDX   #VRMLO     ; now the low byte
0C87  AD 97 0C          LDA   COLADR
0C8A  20 99 0C          JSR   STRVDC
0C8D  A2 1F             LDX   #VRDAT     ; the data to write
0C8F  AD 96 0C          LDA   TEMPA      ; retrieve the value from .A
0C92  20 99 0C          JSR   STRVDC     ;
0C95  60                RTS              ; and it's done
                                         ;
0C96  00        TEMPA   .BYTE 0
0C97  00 00     COLADR  .BYTE 0,0
0C99            STRVDC  =     *          ; .X is the register; .A is the information
0C99  8E 00 D6          STX   VDCADR     ; store the register address
0C9C  AE 00 D6  SVLOOP  LDX   VDCADR     ; now wait
0C9F  10 FB             BPL   SVLOOP     ; until bit 7 is set
0CA1  8D 01 D6          STA   VDCDAT     ; and store the data
0CA4  60                RTS
```

See also CUST80, RE80CO, WR80CO.

VERIFY

Name

Verify a disk file

Description

VERIFY checks a copy of your BASIC or ML program on disk to insure that it is the same as the one currently in memory. If there are any differences between the program in memory and the corresponding one on disk, this routine prints NOT OK.

Prototype

1. On the 128, set the bank to 15.
2. Set the parameters as 1,8,1 to verify "PROGRAM" (SETLFS, SETNAM).
3. On the 128, prior to SETNAM, load .A with the bank containing the program you wish to verify and .X with the bank containing its filename. Then JSR to SETBNK.
4. Store a 1 in .A for to indicate a verify operation.
5. JSR to the Kernal routine LOAD.
6. Check the carry flag for a disk error (carry is set after a LOAD error).
7. Check bit 4 of the I/O status flag at **144** to see if the error was a verify error.
8. If bit 4 is set, print NOT OK.
9. Otherwise, print OK.

Explanation

This routine is very straightforward. To use it, simply substitute for PROGRAM the name of the program you wish to verify.

Notice that **VERIFY** is similar in many ways to the load routines (**LOADAB, LOADBS, LOADRL**). A key difference in the setup is the value placed in the accumulator prior to JSRing to LOAD. A value of zero in .A indicates that a load operation is to be performed. A nonzero value signifies a verify operation.

There are probably several ways to see whether the program in memory has verified properly. A direct way, employed here, is to check bit 4 of the status flag at **144**. If this bit is set, a verify error has occurred and the full error message NOT OK is printed. If bit 4 is cleared, meaning no verify error has occurred, the index pointer to the error message, .Y, is offset in MSGNOK, so only OK gets printed. This trick prevents us from having to include a routine to print the second message.

Note: **VERIFY** currently lacks complete disk error check-

ing (except for checking the carry flag after JSR LOAD). You can add this feature if you like by incorporating the subroutine **DERRCK** into the code. Place **DERRCK** just before FILENM, as noted in the source listing. Jump to **DERRCK** immediately after the JSR LOAD instruction. Also, be sure to open the error channel (15) at the beginning of the program (noted in the source listing).

On the 128, you must define and include BNKNUM and BNKFNM at the end of the program.

Routine

```
C000              SETLFS   =     65466
C000              SETNAM   =     65469
C000              LOAD     =     65493
C000              CHROUT   =     65490
C000              STATUS   =     144
C000              SETBNK   =     65384          ; Kernal bank number for verify and filename
                                                ; (128 only)
C000              MMUREG   =     65280          ; MMU configuration register (128 only)
                                                ;
                                                ; Verify the file (BASIC or ML) on disk.
                                                ;
                                                ; Open channel 15 here if you include disk
                                                ; error checking (DERRCK).
                                                ;
C000              VERIFY   =     *
                                                ; LDA #0; set the 128 to bank 15 (128 only)
                                                ; STA MMUREG; (128 only)
C000  A9 01                LDA   #1             ; logical file number (value doesn't matter)
C002  A2 08                LDX   #8             ; device number for disk drive
C004  A0 01                LDY   #1             ; secondary address of 1 for absolute load
C006  20 BA FF             JSR   SETLFS         ; set parameters for verify
                                                ; Include the following three instructions
                                                ; on the 128 only.
                                                ; LDA BNKNUM; bank containing the
                                                ; program
                                                ; LDX BNKFNM; bank containing the
                                                ; ASCII filename
                                                ; JSR SETBNK
C009  A9 09                LDA   #FNLENG        ; length of filename
C00B  A2 38                LDX   #<FILENM       ; address of filename
C00D  A0 C0                LDY   #>FILENM
C00F  20 BD FF             JSR   SETNAM         ; set up filename
C012  A9 01                LDA   #1             ; flag for verify
C014  20 D5 FF             JSR   LOAD           ; verify the file
                                                ;
                                                ; JSR DERRCK; Insert here for error
                                                ; checking.
                                                ;
C017  08                   PHP                  ; store the carry flag setting
C018  A9 0D                LDA   #13            ; print OK or NOT OK on next line
C01A  20 D2 FF             JSR   CHROUT
C01D  28                   PLP                  ; restore carry flag setting
C01E  B0 0A                BCS   NOTOK          ; if disk error occurs, carry is normally set
                                                ; after load
C020  A5 90                LDA   STATUS         ; check the I/O status flag
C022  29 10                AND   #16            ; test bit 4 for verify error
C024  D0 04                BNE   NOTOK          ; Bit 4 is 1, so verify error occurred. Print
                                                ; "NOT OK".
```

```
C026  A0 04             LDY   #4          ; No verify error. Offset into message to
                                          ; OK.
C028  D0 02             BNE   LOOP        ; print it
C02A  A0 00      NOTOK  LDY   #0
C02C  B9 41 C0   LOOP   LDA   MSGNOK,Y    ; print NOT OK or OK
C02F  F0 06             BEQ   FINISH      ; exit on zero byte
C031  20 D2 FF          JSR   CHROUT
C034  C8                INY
C035  D0 F5             BNE   LOOP        ; continue printing message
C037  60         FINISH RTS
                                          ;
                                          ; Insert DERRCK here if you're including
                                          ; disk error checking.
                                          ;
C038  30 3A 50   FILENM .ASC  "0:PROGRAM" ; Substitute the name of your program here
                                          ; (<=16 characters)
C041             FNLENG =     * - FILENM  ; length of filename
                                          ; Include the next two variables for the 128
                                          ; only
C041  4E 4F 54   MSGNOK .ASC  "NOT OK"    ; message for NOT OK/OK
C047  00                BYTE  0
                                          ; BNKNUM .BYTE 0; bank number where
                                          ; program to verify is located
                                          ; BNKFNM .BYTE 0; bank number where
                                          ; ASCII filename is located
```

See also SAVEBS, SAVEML.

Name

Change the text screen location

Description

This routine lets you change the text screen location within the current video bank. The text screen must be placed on an even 1K boundary within the video bank. The high nybble, bits 4–7, of the VIC-II chip memory control register (53272) determines the actual offset of the text screen within the chosen video bank. This offset can have values from 0 through 15.

Prototype

1. Enter this routine with .A containing the 1K offset for the text screen.
2. Multiply .A by 16, shifting the low nybble to the high nybble.
3. Store the result into bits 4–7 of the VIC-II memory control register.

Explanation

The example routine locates the text screen at 8192, or at the 8K boundary, within the current video bank. Here, SCROFF contains the offset (in 1K increments) to the start of text screen memory. For instance, change the routine to start the text screen at an offset of 4K, store a 4 in SCROFF.

On the 128, the value in location 2604 (VM1) is copied into 53272 during each IRQ interrupt as long as you're working in a portion of the screen containing text. (If you're in bitmap mode, location 2605, or VM2, is copied into 53272.) So, on the 128, define VMCSB as 2604. (Although it's not necessary, you can also change the label to VM1. If you do this, be sure to change it everywhere it's referenced in the program.)

Note: This routine currently uses a zero-page location (251) for temporary storage. Unfortunately, this and other available zero-page locations are heavily used by many other ML routines. If your program requires you to keep this location free, just reserve a labeled byte at the end of your program for storage (for example, TEMPA .BYTE 0) and substitute the chosen label (here, TEMPA) everywhere ZP occurs in the source code.

Routine

```
C000              VMCSB   =      53272    ; 2604 on the 128—VIC II chip memory
                                          ; control register
C000              ZP      =      251
                                          ;
                                          ; Offset text screen by 8K in current video
                                          ; bank.
C000  AD 18  C0           LDA    SCROFF   ; .A contains screen offset
C003  20 07  C0           JSR    VICADR   ; offset text screen
C006  60                  RTS
                                          ;
                                          ; Offset text screen by 1K times .A in current
                                          ; video bank
C007  0A          VICADR  ASL             ; multiply by 16 to position in high nybble
C008  0A                  ASL
C009  0A                  ASL
C00A  0A                  ASL
C00B  85 FB               STA    ZP       ; store temporarily
C00D  AD 18  D0           LDA    VMCSB    ; retain current bits 0-3 of VMCSB
C010  29 0F               AND    #15
C012  05 FB               ORA    ZP       ; OR in bits 4-7
C014  8D 18  D0           STA    VMCSB    ; and store result in control register
C017  60                  RTS
                                          ;
C018  08          SCROFF  .BYTE  8        ; text screen offset by 8K within video bank
```

See also CHOUTP, VIDBNK.

Name

Change the VIC chip video bank

Description

VIDBNK lets you choose the current 16K VIC chip video bank from four possible choices:

Bank 0 (0–16383)
Bank 1 (16384–32767)
Bank 2 (32768–49151)
Bank 3 (49152–65535)

Prototype

1. Enter the routine with .A containing the chosen video bank number (0–3).
2. Set the CIA #2 port A data direction register for output.
3. Store the result of 3 minus the bank number into bits 0–1 of the CIA #2 port A data register.

Explanation

The VIC chip, which is in charge of all video output on the 64 and all 40-column output on the 128, can access only 16K of memory at any one time. This 16K area is called the *video bank*. Within the selected 16K must reside all video-oriented information: sprite shapes, text screen memory, hi-res screen memory, and character shapes. Bank 0 is the default video bank. Because locations 0–16384 are often used for other purposes, it's sometimes useful to use a different video bank.

Routine

```
C000                CI2PRA   =     56576      ; CIA #2 data port register A
C000                C2DDRA   =     56578      ; CIA #2 data direction register A
C000                ZP       =     251
                                              ;
                                              ; Select video bank 2
C000  AD 22 C0               LDA   BNKNUM     ; bank number (0-3)
C003  20 07 C0               JSR   VIDBNK     ; select bank
C006  60                     RTS
                                              ;
                                              ; Select a video bank. Enter with .A
                                              ; containing the chosen bank number
C007  49 03         VIDBNK   EOR   #3         ; effectively, (3 - bank number)
C009  85 FB                  STA   ZP         ; store it temporarily
C00B  AD 02 DD               LDA   C2DDRA     ; set data direction register for output
C00E  09 03                  ORA   #3
C010  8D 02 DD               STA   C2DDRA
                                              ;
C013  AD 00 DD               LDA   CI2PRA     ; take current CI2PRA value
```

```
C016  29  FC                   AND  #252         ; and keep bits 2-7,
C018  05  FB                   ORA  ZP           ; OR with 3 — bank number
C01A  8D  00  DD               STA  CI2PRA       ; reset register
C01D  60                       RTS
                                                 ;
C01E  02             BNKNUM    .BYTE 2           ; bank 2
```

See also CHOUTP, VICADR.

Name

Warm start

Description

The difference between a *cold start* and *warm start* on a computer is basically one of degree. A warm start always has a less severe effect on the machine.

One way to cause a warm start on the 64 or 128 is to JuMP directly to the warm-start routine. A warm start also occurs anytime a BRK instruction (0) is encountered.

On the 64, the result of a warm start is the same as when you press the RUN/STOP and RESTORE keys simultaneously. On both computers, all page 3 RAM vectors are restored to their initial settings. In addition, the character set and the screen are reset to their original condition.

Following the warm-start routine on the 64, you're returned to BASIC. On the 128, you're placed in the monitor. On both machines, the BASIC program remains intact.

Prototype

JuMP to a location containing a zero.

Explanation

To demonstrate the effect of a warm start, the example program initially changes the screen and text colors. When you press B, **WARMST** is executed, causing the screen and text colors to be restored to their default settings. As we've indicated, on the 64, you're left in BASIC. On the 128, you're left in the monitor.

WARMST is the same for both computers. In either case, you JuMP to a location in memory that you know contains a zero. Here, a zero has been placed in memory (in zero page) from within the main program.

Routine

```
C000        ZP      =   251
C000        GETIN   =   65508
C000        CHROUT  =   65490
C000        BGCOL0  =   53281     ; screen background color register 0
C000        COLOR   =   646       ; COLOR = 241 on the 128—foreground
                                  ; color register for text
C000        EXTCOL  =   53280     ; border-color register
C000        LTGREN  =   13
C000        GREEN   =   5
C000        WHITE   =   1
```

```
                                               ;
                                               ; Perform a warm start with B key
C000  A9  00                LDA  #0            ; store a zero byte in zero page
C002  85  FB                STA  ZP
C004  A9  0D      BCKCOL    LDA  #LTGREN       ; set screen background color to light green
C006  8D  21  D0            STA  BGCOL0
C009  A9  05      BORCOL    LDA  #GREEN        ; set border color to green
C00B  8D  20  D0            STA  EXTCOL
C00E  A9  01      TXTCOL    LDA  #WHITE        ; set text color to white
C010  8D  86  02            STA  COLOR
C013  20  E4  FF  LOOP      JSR  GETIN         ; get a character
C016  F0  FB                BEQ  LOOP          ; if no input
C018  20  D2  FF            JSR  CHROUT        ; print it
C01B  C9  42                CMP  #66           ; is it B?
C01D  D0  F4                BNE  LOOP          ; if not, get another key
C01F  4C  22  C0            JMP  WARMST        ; execute warm start
                                               ;
                                               ; WARMST clears the screen and resets
                                               ; default colors.
C022  4C  FB  00  WARMST    JMP  ZP            ; warm start caused by zero byte and RTS
```

See also COLDST.

Name

Sets windows boundaries using escape codes

Description

A very useful feature of the 128 is its built-in windowing capability. As your programs become more sophisticated, you'll find any number of uses for windows—menus, prompts (Y/N), messages, and so forth. This routine shows how to set up a text window on the 128 by using escape codes.

Prototype

1. Enter with the appropriate window dimensions defined by the variables TOPROW, LEFTCL, BTROWO, and RGTOFF at the end of the program. (Note that BTROWO and RGTOFF are offsets from TOPROW and LEFTCL, respectively.)
2. Position the cursor at the top left corner of the window with PLOT.
3. Print an ESC-T for top.
4. Likewise, position the cursor at the bottom right position with PLOT.
5. Print an ESC-B for bottom.

Explanation

To use PLOT to set up the window boundaries, load the X and Y registers with the row and column number of the window border. With PLOT, the rows and columns are numbered, beginning with zero. Possible row values are 0–24; columns can run from 0 through 39 (or 0–79 on the 80-column screen).

After the top corner position has been fixed with PLOT, we load .A with 84 (for ASCII *T*) and print it in the form of an ESC code using the subroutine ESCPRT. In ESCPRT, the character to be printed is stored on the stack while an ESC code—CHR$(27)—is printed. Following this, we pull the character back off the stack and print it as well.

A similar process is followed in establishing the bottom border of the window. This time an ESC B, which sets the bottom of the window, is printed. It should be noted that the previous action (printing ESC-T) has put the top of the window at a given location and that the Kernal PLOT routine operates relative to the current window. Thus, the values for the bottom of the window are the width and height of the window, *not* the absolute screen coordinates of the bottom corner.

Finally, to clearly show the limits of the window, a continuous stream of *W*'s is printed.

Routine

```
0C00                PLOT     =     65520
0C00                CHROUT   =     65490
                                                  ;
                                                  ; Position window and print W's.
0C00  20  0B  0C             JSR   WINDOW         ; set up the window
0C03  A9  57                 LDA   #87            ; print W
0C05  20  D2  FF  LOOP       JSR   CHROUT
0C08  4C  05  0C             JMP   LOOP           ; again and again and again ..
                                                  ;
                                                  ; Set up a window on the 128 screen.
0C0B  AE  30  0C  WINDOW     LDX   TOPROW         ; top left position
0C0E  AC  31  0C             LDY   LEFTCL
0C11  18                     CLC                  ; clear carry to set position
0C12  20  F0  FF             JSR   PLOT           ; set cursor at .Y,.X
0C15  A9  54                 LDA   #84            ; T for top of window
0C17  20  26  0C             JSR   ESCPRT         ; print ESC-T
0C1A  AE  32  0C             LDX   BTROWO         ; bottom right
0C1D  AC  33  0C             LDY   RGTOFF
0C20  18                     CLC                  ; set position
0C21  20  F0  FF             JSR   PLOT           ; set cursor at .Y,.X
0C24  A9  42                 LDA   #66            ; now print ESC-B for bottom of window
0C26  48          ESCPRT     PHA                  ; save character to print to the stack
0C27  A9  1B                 LDA   #27            ; print ESC
0C29  20  D2  FF             JSR   CHROUT
0C2C  68                     PLA                  ; pull character from stack
0C2D  4C  D2  FF             JMP   CHROUT         ; print it and RTS
                                                  ;
0C30  08          TOPROW     .BYTE 8              ; window's top left corner is on the ninth row
0C31  0A          LEFTCL     .BYTE 10             ; and eleventh column
                                                  ; The following two values are offsets from
                                                  ; the first two.
0C32  08          BTROWO     .BYTE 8              ; window's bottom right corner is on the
                                                  ; seventeenth row
0C33  14          RGTOFF     .BYTE 20             ; and thirty-first column
```

See also BIGMAP.

Name

Open a disk buffer and write a sector to disk

Description

This routine copies a block of 256 bytes from computer memory to a memory buffer inside the disk drive. This is relatively low-level disk output; most of the time you can just read and write program or sequential files. There are times, however, when you will want to write directly to a disk sector (a disk editor must be able to do this and so must a program that "unscratches" a file that's been accidentally scratched).

Prototype

1. OPEN 15,8,15 with no filename (SETLFS, SETNAM, and OPEN).
2. OPEN 1,8,3 with the name # (SETLFS, SETNAM, and OPEN, again). This sets aside a buffer in disk drive memory.
3. Write 256 bytes to logical file 1, the buffer (B-P is optional).
4. Send the U2 (block-write) command to logical file 15 to transfer the buffer to disk.
5. Close open channels.

Explanation

This routine depends heavily on Kernal routines; note the numerous equates at the top of the program. The first JSR goes to the subroutine OPEN15, which opens the command channel to the disk and is the equivalent of the BASIC command OPEN 15,8,15. The usual SETLFS, SETNAM, and OPEN Kernal routines are called. The next subroutine opens logical file 1 (with secondary address of 3) and reserves a buffer by using the special filename #. At address $C006, CHKOUT sets up the buffer to receive output. Finally, 256 bytes are printed to the disk buffer via CHROUT.

Now that our message is in the disk buffer, we have to write it from disk memory to the disk itself. Again CHKOUT diverts output, but to the command channel 15 this time. The command we send (an ASCII string at the end of the program) is *U2 3 0 2 2*. The *U2* means write a block; *3* is the secondary address of the buffer channel, *not* the logical file number. We opened the file as 1,8,3 and printed to 1, but when the memory is copied, we provide the secondary address (channel 3) instead of 1. The next number (ASCII 0) is always a zero, un-

less you happen to own a dual drive. The next two numbers are the track and sector (in that order).

Note: If there's a specific byte or two you'd like to change on a specific sector, you should first read the sector into the disk buffer. Then set the buffer pointer with the B-P command, write the character, and copy memory back to the appropriate sector.

Warning: This routine writes directly to a disk sector, regardless of what might already be there. If you're going to experiment with this routine, *don't use a disk that contains important files.* If you write information to disk sectors, they may be overwritten by later disk access, unless you mark the sector as allocated in the BAM.

Routine

```
C000                   SETLFS   =    $FFBA
C000                   SETNAM   =    $FFBD
C000                   OPEN     =    $FFC0
C000                   CHKOUT   =    $FFC9
C000                   CHKIN    =    $FFC6
C000                   CHROUT   =    $FFD2
C000                   CHRIN    =    $FFCF
C000                   CLOSE    =    $FFC3
C000                   CLRCHN   =    $FFCC
                                               ;
C000  20 47 C0  WRBUFF  JSR     OPEN15         ; open command channel
C003  20 5E C0          JSR     OPNBUF         ; open a buffer
C006  A2 01             LDX     #1
C008  20 C9 FF          JSR     CHKOUT         ; ready to send to channel 1 (the buffer)
C00B  90 03             BCC     BUFOK          ; carry clear if no error
C00D  4C 79 C0          JMP     ERROR          ; else print error message
C010  A0 00     BUFOK   LDY     #0             ; index = 0
C012  B9 BB C0  WRITE   LDA     BLOCK,Y        ; start writing to the buffer
C015  20 D2 FF          JSR     CHROUT         ; send it out
C018  C8                INY                    ; increment index
C019  D0 F7             BNE     WRITE          ; and go back for another, until 256 bytes
                                               ; are sent
C01B  20 CC FF          JSR     CLRCHN         ; back to normal I/O
                                               ;
C01E  A2 0F             LDX     #15            ; open the command channel
C020  20 C9 FF          JSR     CHKOUT         ; for output
C023  90 03             BCC     OUTOK          ; carry clear = OK
C025  4C 79 C0          JMP     ERROR          ; otherwise, an error
C028  A0 00     OUTOK   LDY     #0             ; start counter at zero
C02A  B9 8E C0  SENDIT  LDA     BLKWR,Y        ; get a character
C02D  F0 07             BEQ     QUIT15
C02F  20 D2 FF          JSR     CHROUT         ; and send it
C032  C8                INY                    ; count up
C033  4C 2A C0          JMP     SENDIT         ; and continue
C036  20 CC FF  QUIT15  JSR     CLRCHN         ; restore I/O
                                               ;
                                               ; All done, so close it down.
C039  A9 01     FINIS   LDA     #1
C03B  20 C3 FF          JSR     CLOSE          ; close logical file 1
C03E  A9 0F             LDA     #15
C040  20 C3 FF          JSR     CLOSE          ; and the command channel
C043  20 CC FF          JSR     CLRCHN         ; and clear the channels
```

```
C046  60                     RTS              ; done
                                              ; subroutines
C047  A9  0F        OPEN15   LDA   #15        ; logical file number
C049  A2  08                 LDX   #8         ; device number for disk drive
C04B  A0  0F                 LDY   #15        ; secondary address for command channel
C04D  20  BA  FF             JSR   SETLFS     ; set parameters to be opened
C050  A9  00                 LDA   #0         ; length is zero (no filename)
C052  20  BD  FF             JSR   SETNAM
C055  20  C0  FF             JSR   OPEN       ; open it
C058  90  03                 BCC   OK15       ; check for error
C05A  4C  79  C0             JMP   ERROR      ; print the message if there's a problem
C05D  60            OK15     RTS              ; and we're done
                                              ;
                                              ; OPNBUF opens a disk buffer for writing.
C05E  A9  01        OPNBUF   LDA   #1         ; logical file number
C060  A2  08                 LDX   #8         ; device number for disk drive
C062  A0  03                 LDY   #3         ; secondary address for buffer channel
C064  20  BA  FF             JSR   SETLFS
C067  A9  01                 LDA   #1         ; one character
C069  A2  8D                 LDX   #<BUFNAM   ; the filename is "#"
C06B  A0  C0                 LDY   #>BUFNAM
C06D  20  BD  FF             JSR   SETNAM     ; set up the name
C070  20  C0  FF             JSR   OPEN       ; now it's ready
C073  90  03                 BCC   OKBUF      ; to OKBUF if no error
C075  4C  79  C0             JMP   ERROR      ; JMP to error if there is
C078  60            OKBUF    RTS              ; and we're done
                                              ;
C079  20  CC  FF    ERROR    JSR   CLRCHN     ; close down and clear channels
C07C  A0  00                 LDY   #0         ; ready to print message
C07E  B9  9A  C0    MORE     LDA   ERRMSG,Y
C081  F0  07                 BEQ   MSGEND     ; message ends with zero
C083  20  D2  FF             JSR   CHROUT     ; print the character
C086  C8                     INY              ; increment the index
C087  4C  7E  C0             JMP   MORE       ; and go back
C08A  4C  39  C0    MSGEND   JMP   FINIS      ; finish closing files
                                              ;
                                              ; variables
C08D  23            BUFNAM   .ASC  "#"
C08E  55  32  20    BLKWR    .ASC  "U2 3 0 2 2"
                                              ; U2 is block write command
                                              ; 3 is secondary address
                                              ; 0 is drive number
                                              ; track 2, sector 2.
C098  0D  00                 .BYTE 13, 0
C09A  53  4F  4D    ERRMSG   .ASC  "Something is wrong with the disk"
C0BA  00                     .BYTE 0
C0BB                BLOCK    =     *
C0BB  54  48  49             .ASC  "this is a string"
C0CB  20  57  48             .ASC  " which we are writing"
C0E0  20  54  4F             .ASC  " to the disk at track 2"
C0F7  20  53  45             .ASC  " sector 2"
C100  0D                     .BYTE 13
```

See also RDBUFF.

Name

Write a buffer to a sequential or program file

Description

WRITBF relies on three file-handling routines—specifically, **OPENFL**, **WRITFL**, and **CLOSFL** —to write a data buffer to disk. This buffer, whose address is in zero page, can be written as either a sequential or a program file.

Prototype

In the calling program (MAIN, below):

1. Define the length of the data buffer to write to disk (as LENGTH).
2. On the 128, set the bank to 15. On both machines, store the address of the data buffer in zero page. Then place the buffer length in the .X and .Y registers (low byte in .X, high byte in .Y). Finally, JSR to **WRITBF**.

In **WRITBF** itself:

3. Store the buffer length, in .X and .Y upon entry, into a two-byte address (here, BUFCTR).
4. Open a sequential or program filename with **OPENFL**.
5. Write the data buffer to the open file with **WRITFL**.
6. Close the open file with **CLOSFL**.

Explanation

In the example program, we use **WRITBF** to write the buffer containing the message FILE SEQUENTIAL IS 37 CHARACTERS LONG to disk as a sequential file. See **WRITFL** for an explanation of how to write a program file.

Although it may look like a long routine, **WRITBF** is very short. The buffer length that is in .X (low byte) and .Y (high byte) upon entry is immediately stored in BUFCTR. From this point on, it's just a matter of accessing the three routines described elsewhere in this book.

Note: You can add disk error checking to this program by including **DERRCK**, as we've done for several other disk-related routines in this book.

Routine

```
C000          SETLFS   =   65466
C000          SETNAM   =   65469
C000          OPEN     =   65472
C000          CHKOUT   =   65481
C000          CHROUT   =   65490
C000          CLOSE    =   65475
```

```
C000                  CLRCHN   =   65484
C000                  ZP       =   251
C000                  SETBNK   =   65384      ; Kernal bank number for data and filename
                                              ; (128 only)
C000                  MMUREG   =   65280      ; MMU configuration register (128 only)
                                              ;
                                              ; WRITBF uses the following three routines
                                              ; to write characters
                                              ; from a buffer in memory to a sequential or
                                              ; program file:
C000
                                              ; OPENFL to open the sequential/program
                                              ; file,
                                              ; WRITFL to write characters to the file, and
                                              ; CLOSFL to close the file and restore
                                              ; the default output device.
                                              ;
                                              ; Enter WRITBF with buffer address in zero
                                              ; page, length in .X, .Y.
C000                  MAIN     =   *
                                              ; LDA #0; set the 128 to bank 15 (128 only)
                                              ; STA MMUREG; (128 only)
C000  A9 71                    LDA #<BUFFER   ; store address of buffer to zero page
C002  85 FB                    STA ZP
C004  A0 C0                    LDY #>BUFFER
C006  84 FC                    STY ZP+1
C008  AE 96 C0                 LDX LENGTH     ; store length of buffer in .X (low) and .Y
                                              ; (high)
C00B  AC 97 C0                 LDY LENGTH+1
C00E  20 12 C0                 JSR WRITBF     ; go write data to file
C011  60                       RTS
                                              ;
                                              ; WRITBF opens a SEQ or PRG file data
                                              ; from buffer at ZP.
                                              ; Enter the routine with buffer length in .X
                                              ; (low byte) and .Y (high).
C012  8E 98 C0  WRITBF         STX BUFCTR     ; store length of buffer (in .X and .Y) to
                                              ; memory
C015  8C 99 C0                 STY BUFCTR+1
C018  20 23 C0                 JSR OPENFL     ; OPEN the file with parameters 1,8,2
C01B  20 39 C0                 JSR WRITFL     ; write data from buffer to open file
C01E  A9 01                    LDA #1         ; file to close
C020  4C 58 C0                 JMP CLOSFL     ; close file 1, restore default devices, and
                                              ; return to MAIN
                                              ;
                                              ; OPENFL opens a sequential or program file
                                              ; with 1,8,2 for reading or writing.
C023                  OPENFL   =   *
                                              ; Open channel 15 here if you include error
                                              ; checking (DERRCK).
                                              ;
C023  A9 01                    LDA #1         ; logical file 1
C025  A2 08                    LDX #8         ; device number for disk drive
C027  A0 02                    LDY #2         ; secondary address (2-14 are OK)
C029  20 BA FF                 JSR SETLFS     ; file parameters set
                                              ; Include the following three instructions on
                                              ; the 128 only
                                              ; LDA BNKNUM; bank number for file data
                                              ; LDX BNKFNM; bank number for ASCII
                                              ; filename
                                              ; JSR SETBNK
                                              ;
C02C  A9 10                    LDA #FNLENG    ; length of filename
C02E  A2 61                    LDX #<FILENM   ; address of filename
```

```
C030  A0 C0              LDY   #>FILENM
C032  20 BD FF           JSR   SETNAM       ; set up filename
                                            ;
C035  20 C0 FF           JSR   OPEN         ; open the file for writing
                                            ;
                                            ; JSR DERRCK; Insert here for disk error
                                            ; checking
                                            ;
C038  60                 RTS                ; return to WRITBF
                                            ;
                                            ; WRITFL writes characters from a buffer
                                            ; whose address is in zero page
                                            ; to a sequential or program file.
C039  A2 01     WRITFL   LDX   #1
C03B  20 C9 FF           JSR   CHKOUT       ; send output to file 1
                                            ;
                                            ; Include the following four lines to send the
                                            ; load address for a program file.
                                            ; LDA ZP; output low/high byte address of
                                            ; buffer in zero page to disk
                                            ; JSR CHROUT
                                            ; LDA ZP+1
                                            ; JSR CHROUT
                                            ;
C03E  A0 00              LDY   #0           ; initialize index into the storage buffer
C040  B1 FB     WRLOOP   LDA   (ZP),Y       ; load a character from buffer
C042  20 D2 FF           JSR   CHROUT       ; send it to the open file
C045  E6 FB              INC   ZP           ; increment low byte of buffer address
C047  D0 02              BNE   LENCHK       ; low byte hasn't turned over, so skip forward
C049  E6 FC              INC   ZP+1         ; otherwise, increase high byte
C04B  CE 98 C0  LENCHK   DEC   BUFCTR       ; decrement low byte of buffer counter
C04E  D0 F0              BNE   WRLOOP       ; if not equal, more of the buffer remains, so
                                            ; continue writing
C050  CE 99 C0           DEC   BUFCTR+1     ; otherwise, decrement the high byte of
                                            ; buffer counter
C053  AD 99 C0           LDA   BUFCTR+1     ; continue writing until last page of buffer
                                            ; has been sent
C056  C9 FF              CMP   #255         ; high byte goes from 0 through 255 on last
                                            ; page
C058  D0 E6              BNE   WRLOOP       ; we've yet to reach last page, so write on
C05A  60                 RTS                ; return to WRITBF
                                            ;
                                            ; CLOSFL closes the logical file in .A and
                                            ; restores default devices.
C05B  20 C3 FF  CLOSFL   JSR   CLOSE        ; close file in .A
C05E  4C CC FF           JMP   CLRCHN       ; clear all channels, restore default devices,
                                            ; and RTS
                                            ;
                                            ; Insert DERRCK routine here if you're
                                            ; including error checking.
                                            ;
C061  30 3A 53  FILENM   .ASC  "0:SEQUENTIAL,S,W"
                                            ; example sequential file to write
                                            ; ,S,W is optional with sequential file writes.
                                            ; Change filename to "0:PROGRAM,P,W" to
                                            ; write a program file.
C071            FNLENG   =     *-FILENM     ; length of filename
                                            ;
```

```
C071  46  49  4C  BUFFER   .ASC    "FILE SEQUENTIAL IS 37 CHARACTERS LONG"
C096  25  00      LENGTH   .WORD37           ; two bytes for storing buffer length
C098  00  00      BUFCTR   .WORD0            ; two-byte counter for remaining number of
                                             ; bytes to write
                                             ;
                                             ; Include the next two variables on the 128.
                                             ; BNKNUM  BYTE 0; bank number for file
                                             ; data
                                             ; BNKFNM  BYTE 0; bank number where
                                             ; ASCII filename is located
```

See also CLOSFL, WRITFL.

Name

Send characters to a sequential or program file

Description

This routine transfers data from a buffer whose address is in zero page to an open file. It's intended to be used in a program where sequential or program data is written to disk, such as **WRITBF**.

Prototype

1. Before accessing **WRITFL**, call **OPENFL** to open a channel where the data will be written. Also, store the address of the data buffer to be written to disk in zero page.
2. Define the output channel as the one opened with CHKOUT.
3. If you're outputting a program file and wish to include a program load address, send the buffer address bytes (low byte first, then high byte) stored in zero page.
4. Write a given number of bytes (the number is stored in the counter BUFCTR) from the buffer to the open channel. Then return to the calling program.

Explanation

WRITFL takes data from a buffer and outputs it to an open disk file until BUFCTR bytes have been sent. The routine assumes the data buffer's address is in zero page (in $FB, labeled ZP). Be sure to set up this pointer before accessing **WRITFL**.

In the example below, the logical file used for the transfer is 1. This file number should have been assigned previously by a routine that opened the data channel. If you need to output data through some other channel, such as the printer or tape drive, load the X register with the appropriate value in $C000.

Routine

```
C000                CHKOUT  =     65481
C000                CHROUT  =     65490
C000                ZP      =     251

                                           ; WRITFL writes characters from a buffer
                                           ; whose address is in zero page
                                           ; to a sequential or program file.
C000  A2  01        WRITFL  LDX   #1
C002  20  C9  FF            JSR   CHKOUT   ; send output to file 1
                                           ;
                                           ; Include the following four lines to send
                                           ; the load address.
                                           ; LDA ZP; output low/high byte address of
                                           ; buffer in zero page to disk
```

```
                                                  ; JSR CHROUT
                                                  ; LDA ZP+1
                                                  ; JSR CHROUT
                                                  ;
C005  A0 00              LDY   #0                 ; initialize index into the storage buffer
C007  B1 FB      WRLOOP  LDA   (ZP),Y             ; load a character from buffer whose
                                                  ; address is in ZP
C009  20 D2 FF           JSR   CHROUT             ; send it to the open file
C00C  E6 FB              INC   ZP                 ; increment low byte of buffer address
C00E  D0 02              BNE   LENCHK             ; low byte hasn't rolled over, so skip
                                                  ; forward
C010  E6 FC              INC   ZP+1               ; otherwise, increase high byte
C012  CE 22 C0   LENCHK  DEC   BUFCTR             ; decrement low byte of buffer counter
C015  D0 F0              BNE   WRLOOP             ; if not equal, more of the buffer remains,
                                                  ; so continue writing
C017  CE 23 C0           DEC   BUFCTR+1           ; otherwise, decrement the high byte of
                                                  ; buffer counter
C01A  AD 23 C0           LDA   BUFCTR+1           ; continue writing until last page of buffer
                                                  ; has been sent
C01D  C9 FF              CMP   #255               ; high byte goes from 0 to 255 on last page
C01F  D0 E6              BNE   WRLOOP             ; we've yet to reach last page, so write on
C021  60                 RTS                      ; return to main program
                                                  ;
C022  00 00      BUFCTR   WORD0                   ; two-byte counter for remaining number of
                                                  ; bytes to write
```

See also CLOSFL, WRITBF.

Name

Set colors for extended background color mode

Description

Extended background color mode reduces the size of the available character set from 256 characters to only 64. But at the same time, you have a choice of four different background colors, with no loss of horizontal resolution. This routine sets the four background colors.

Prototype

Read the four color values from EXBCOL and store them beginning at location 53281 (BGCOL0).

Explanation

To set the background colors, assign the color values for the four groups of characters (0–63, 64–127, 128–191, and 192–255) in EXBCOL at the end of the program.

The program fragment below illustrates how the four colors are set. For a complete example of extended background color mode, see **XBCMOD**.

Routine

```
C000                BGCOL0   =     53281        ; text background color register 0
                                                ;
C000  A2 03         XBCCOL   LDX   #3           ; as an index
C002  BD 0C  C0     COLOOP   LDA   EXBCOL,X     ; get each color value
C005  9D 21  D0              STA   BGCOL0,X     ; assign it to a register
C008  CA                     DEX                ; for next register
C009  10 F7                  BPL   COLOOP       ; do all four
C00B  60                     RTS
                                                ;
C00C  03 04  05     EXBCOL   .BYTE 3,4,5,2      ; colors—cyan, purple, green, red
```

See also: XBCMOD, MTCCOL, MTCMOD.

Name

Turn extended background color mode on or off

Description

Two closely related routines are demonstrated here in one program. The first routine, **XBCMOD**, turns on (or off) extended background color mode while the second, **XBCCOL**, sets the colors for this mode.

By using these two routines in your programs, some interesting special effects can be achieved.

Prototype

1. Load the contents of the vertical fine-scrolling/control register at 53265 (SCROLY) into the accumulator.
2. ORA with %01000000 to turn on bit 6 and store the result back into the register. (To turn off extended background color mode, AND the contents of SCROLY with %10111111.)

Explanation

Normally, the background color for text characters is taken from the color register at 53281, or BGCOL0. But by activating extended background color mode (setting bit 6 of SCROLY), each character's background color is instead taken from one of four color registers (53281–53284), depending on the screen code of the character to be displayed.

In this mode, the screen codes are divided into four groups: 0–63, 64–127, 128–191, and 192–255. Only characters from the first group (screen codes 0–63) can be displayed. Fortunately, this group contains most of the characters you ordinarily need (you may wish to define new characters if you'd rather use 64 other characters). Within this group are the letters *A–Z*, the numbers 0–9, and the punctuation marks. When one of the characters from this group is printed, the background color for the character is taken from BGCOL0.

Characters with screen codes above 63 will appear the same as the first group (screen codes 0–63), except that their background colors will come from one of the three remaining color registers (53282–53284). To determine what a particular character will look like on the screen if its display code is higher than 63, subtract the initial screen code for the group from the intended display code and locate the corresponding character in the first group of screen codes.

For example, if you placed the spade character (screen code 65) on the screen, and turned on extended background

color mode, you'd see the letter *A* (screen code 65 64 = 1) in a background color taken from the register at 53282 (BGCOL1).

The fact that each group of screen codes has a different background color in this mode allows you to create some impressive animation and windowing effects. For instance, if you place characters from each of the four screen-code groups on the screen at once, and cycle the color values in each group's color register, a three-dimensional movement effect can be achieved. You can also simulate a window by printing certain messages using characters from just one screen-code group. These effects, of course, take on an added dimension if you use redefined characters.

Take a look at the example program to see how these two routines work together. In SCRLOP, we first display all screen codes (0–255) at the top of the screen. When you press a key, extended background color mode is activated with **XBCMOD**, and the respective colors for the four groups of screen codes are assigned in **XBCCOL**. The result is that the first 64 screen codes are now displayed four times. And each group of screen codes is shown in a different background color.

Note: While in extended background color mode, if you need a character not available in the first 64 screen codes, you'll have to define it yourself. You can perform this task with a character-redefinition routine like **CHRDEF**.

Routine

```
C000                BGCOL0   =    53281       ; text background color register 0
C000                SCROLY   =    53265       ; scroll/control register
C000                SCREEN   =    1024
C000                CHROUT   =    65490
C000                GETIN    =    65508
                                              ;
                                              ; Display screen codes 0–255. Then turn on
                                              ; extended background color mode,
                                              ; set extended background colors, and again
                                              ; display screen codes 0–255.
C000  20  03  C0  MAIN     JSR  CHRCLR        ; clear screen, display 0–255 screen codes,
                                              ; and wait for key
                                              ;
                                              ; Clear the screen and display 0–255 screen
                                              ; codes.
C003  A9  93      CHRCLR   LDA  #147          ; clear the screen
C005  20  D2  FF           JSR  CHROUT
C008  A0  00               LDY  #0            ; as an index in SCRLOP
C00A  98          SCRLOP   TYA
C00B  99  00  04           STA  SCREEN,Y      ; display 0–255 screen codes in normal mode
C00E  C8                   INY                ; for next screen code
C00F  D0  F9               BNE  SCRLOP        ; and continue
C011  20  E4  FF  GETKEY   JSR  GETIN         ; wait for a keypress
C014  F0  FB               BEQ  GETKEY        ; if no keypress, then wait
```

```
C016  20 1C C0          JSR   XBCMOD      ; turn on extended background color mode
C019  4C 25 C0          JMP   XBCCOL      ; assign extended background colors and RTS
                                          ;
                                          ; Turn on (or off) extended background color
                                          ; mode.
C01C  AD 11 D0  XBCMOD  LDA   SCROLY      ; get current register value
C01F  09 40             ORA   #%01000000  ; turn on bit 6 (turn off with AND
                                          ; %10111111 here)
C021  8D 11 D0          STA   SCROLY      ; and set the register
C024  60                RTS
                                          ;
                                          ; Assign 4 colors to extended background
                                          ; color registers 53281-53284.
C025  A2 03     XBCCOL  LDX   #3          ; as an index
C027  BD 31 C0  COLOOP  LDA   EXBCOL,X    ; get each color value
C02A  9D 21 D0          STA   BGCOL0,X    ; assign it to a register
C02D  CA                DEX               ; for next register
C02E  10 F7             BPL   COLOOP      ; do all four
C030  60                RTS
                                          ;
C031  03 04 05  EXBCOL  .BYTE 3,4,5,2     ; colors—cyan, purple, green, red
```

See also XBCCOL, MTCCOL, MTCMOD.

Index by Topic

Index by Label

To order your copy of *COMPUTE!'s Machine Language Routines for the Commodore 64 and 1284 Disk*, call our toll-free US order line: 1-800-346-6767 (in NY 212-887-8525) or send your prepaid order to:

COMPUTE!'s Machine Language Routines for the Commodore 64 and 128 Disk
COMPUTE! Publications
P.O. Box 5038
F.D.R. Station
New York, NY 10150

All orders must be prepaid (check, charge, or money order). NC residents add 5% sales tax. NY residents add 8.25% sales tax.

Send _____ copies of *COMPUTE!'s Machine Language Routines for the Commodore 64 and 128 Disk* at $12.95 per copy. (858BDSK)

Subtotal $__________

Shipping and Handling: $2.00/disk $__________

Sales tax (if applicable) $__________

Total payment enclosed $__________

□ Payment enclosed
□ Charge □ Visa □ MasterCard □ American Express

Acct. No. ______________________________ Exp. Date ______ (Required)

Name ____

Address ____

City _________________________ State ______ Zip ______

Please allow 4-5 weeks for delivery.

The Machine Language Library

If you're interested in machine language programming on the Commodore 64 or 128, this book is a necessity. *Machine Language Routines for the Commodore 64 and 128* gives programmers a library of ML routines for the 64 and 128 personal computers. In an easy-to-use dictionary arrangement, it puts over 200 indispensable routines at your fingertips. Each routine is fully described and is accompanied by an example program that demonstrates its use. As a bonus, the routines are ready to be plugged into your own programs. Cross-references direct you to other routines that perform similar or related functions.

Here's a sample of what you'll find inside for the beginner:

- Numerous short routines that perform mathematical functions.
- Routines that explain how to read, write, and manipulate disk files.
- Programs to convert strings and numbers.
- Easy-to-use techniques for reading joysticks and for adding sound effects and music to your programs.

And for the more advanced programmer:

- Interrupt-driven programs for playing music.
- Routines to move sprites automatically.
- Programs to display 16 sprites at the same time.
- Examples of how to pass values between ML and BASIC.
- And much more.

Authors Todd Heimarck and Patrick Parrish have combined a wealth of knowledge and experience to create this information-packed sourcebook. With clear explanations and useful examples, *Machine Language Routines for the Commodore 64 and 128* is a handy reference guide as well as an exceptional tutorial.

The source code for each of the routines in this book is also available on a companion disk. See the coupon in the back of the book for details.

ISBN 0-87455-085-8